Group Factorizations and Cryptology

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Dominik Reichl

aus Reutlingen

Tübingen
2015

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	23.07.2015
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Peter Hauck
2. Berichterstatter:	Prof. Dr. Klaus Reinhardt

Zusammenfassung

Asymmetrische Kryptosysteme, auch Public-Key-Systeme genannt, können u.a. zur Verschlüsselung von Daten, Authentifizierung und Sicherstellung der Integrität von Daten eingesetzt werden. Solche Systeme sind in zahlreichen Protokollen zu finden, z.B. in den Bereichen World Wide Web (HTTPS basierend auf SSL/TLS), E-Mail (S/MIME, OpenPGP/PGP), entfernte Befehlsausführung (SSH), Dateitransfer (SCP), und vielen weiteren.

Einwegfunktionen sind Funktionen, die sich effizient berechnen lassen, aber sehr schwierig zu invertieren sind. Einwegfunktionen, die sich mit einer Zusatzinformation (dem privaten Schlüssel) doch effizient invertieren lassen, werden Falltürfunktionen genannt. Asymmetrische Kryptosysteme basieren auf Falltürfunktionen. Die meisten der heute verwendeten asymmetrischen Verfahren, insbesondere RSA und ElGamal, basieren auf solchen Funktionen in kommutativen algebraischen Strukturen. Ob kommutative Strukturen aufgrund deren Eigenschaften ein Sicherheitsproblem darstellen könnten, lässt sich derzeit nicht sagen. Auf jeden Fall ist die Entwicklung und Untersuchung von Verfahren, die auf nicht-kommutativen Strukturen beruhen, sinnvoll.

Ein Kryptosystem namens MST_1, das von S. S. Magliveras, D. R. Stinson und T. van Trung vorgestellt wurde, basiert auf sogenannten logarithmischen Signaturen von beliebigen (also auch nicht-kommutativen, d.h. nicht-abelschen) endlichen Gruppen. Für eine endliche Gruppe G wird eine geordnete Menge α von Teilmengen von G betrachtet, wobei jedes Element von G eine eindeutige Darstellung als Produkt von jeweils einem Element aus den Teilmengen in α haben soll. α wird dann eine logarithmische Signatur genannt. Interessant ist nun die Frage, ob es logarithmische Signaturen gibt, für die es schwierig ist, für ein gegebenes Gruppenelement eine solche Faktorisierung als Produkt in α zu finden. Falls ja, dann wäre dies eine Einwegfunktion: Produkte von Gruppenelementen (mit Faktoren aus α) können effizient berechnet werden, aber die Invertierung, d.h. das Finden einer Faktorisierung als Produkt in α, wäre schwierig.

In dieser Dissertation wird für verschiedene Gruppen und Typen logarithmischer Signaturen die Realisierbarkeit und Sicherheit von MST_1 untersucht.

Der erste Teil der Dissertation befasst sich mit der Erzeugung von logarithmischen Signaturen. Logarithmische Signaturen effizient erzeugen zu können ist eine Voraussetzung für eine konkrete Realisierung des MST_1-Kryptosystems.

Wir untersuchen zunächst Transformationen logarithmischer Signaturen (deren Effekt auf Faktorisierungsabbildungen, Unterklassen von Transformationen, Hintereinanderausführungen von Transformationen, usw.). Basierend darauf entwickeln wir einen Algorithmus zur Erzeugung logarithmischer Signaturen. Dieser funktioniert auch mit nicht-abelschen Gruppen, und bei abelschen Gruppen werden in der Regel mehr logarithmische Signaturen erzeugt als bei den üblicherweise in der Literatur verwendeten Verfahren.

Danach betrachten wir das Faktorisierungsproblem bzgl. logarithmischer Signaturen für verschiedene Gruppen.

Für abelsche Gruppen entwickeln wir Faktorisierungsalgorithmen, die bei bestimmten Klassen von logarithmischen Signaturen effizient sind. Außerdem entwickeln wir einen generischen Faktorisierungsalgorithmus, der nicht nur mit logarithmischen Signaturen sondern mit allen Blocksequenzen funktioniert (wobei die Laufzeit von der Struktur der Eingabe abhängig ist), und bei dem für einige große Klassen logarithmischer Signaturen die Effizienz gezeigt werden kann.

Des Weiteren untersuchen wir logarithmische Signaturen von Diedergruppen, und geben effiziente Faktorisierungsalgorithmen für bestimmte Typen logarithmischer Signaturen an.

Diese Ergebnisse werden verallgemeinert und ausgebaut; wir untersuchen u.a. die verallgemeinerte Quaternionengruppe und Kranzprodukte.

Bei den vorherigen Untersuchungen wurde jeweils eine bestimmte Darstellung der Gruppe verwendet. In einem weiteren Teil der Dissertation analysieren wir, in welchen Fällen sich für eine beliebig gegebene Gruppe (Black-Box-Gruppe) mit bekannter Struktur ein effizienter Algorithmus zur Konvertierung von Gruppenelementen in die Darstellung, die in den vorherigen Kapiteln verwendet wurde, angeben lässt.

Abschließend stellen wir unser Programm vor, in dem ein Kryptosystem (basierend auf einem verallgemeinerten MST_1) und die verschiedenen in dieser Arbeit entwickelten Erzeugungs- und Faktorisierungsalgorithmen implementiert wurden.

Summary

Asymmetric cryptosystems, also called public-key systems, can for instance be used for encrypting data, authentication and integrity checking of data. Such systems can be found in numerous protocols, e.g. in the areas World Wide Web (HTTPS based on SSL/TLS), e-mail (S/MIME, OpenPGP/PGP), remote command execution (SSH), file transfer (SCP), and many more.

One-way functions are functions that can be computed efficiently, but are hard to invert. One-way functions that can be inverted efficiently with an additional information (the private key) are called trap-door functions. Asymmetric cryptosystems are based on trap-door functions. Most of the currently used asymmetric systems, especially RSA and ElGamal, are based on such functions in commutative algebraic structures. Whether commutative structures are a security issue due to their properties is unknown up to now. In any case the development and analysis of systems that are based on non-commutative structures is reasonable.

A cryptosystem called MST_1, which has been introduced by S. S. Magliveras, D. R. Stinson and T. van Trung, is based on so-called logarithmic signatures of arbitrary (also including non-commutative, i.e. non-abelian) finite groups. For a finite group G an ordered set α of subsets of G is being regarded, where each element of G has a unique representation as product of one element from each subset in α. α is then called a logarithmic signature. An interesting question now is whether there exist logarithmic signatures where it is hard to find a factorization as product in α for a given group element. If yes, this would be a one-way function: products of group elements (with factors from α) can be computed efficiently, but the inversion, i.e. finding a factorization as product in α, would be hard.

In this dissertation the realizability and security of MST_1 is analyzed for various groups and logarithmic signature types.

The first part of the dissertation deals with the generation of logarithmic signatures. The ability to efficiently generate logarithmic signatures is a requirement for a concrete realization of the MST_1 cryptosystem.

We first investigate transformations of logarithmic signatures (their effect on factorization mappings, subclasses of transformations, compositions of transformations, etc.). Based on this, we develop an algorithm for generating logarithmic signatures. This algorithm also works with non-abelian groups, and for abelian groups the set of generated logarithmic signatures is typically a proper superset of the logarithmic signatures generated by the methods usually used in literature.

Subsequently, we regard the factorization problem with respect to logarithmic signatures for various groups.

For abelian groups we develop factorization algorithms that are efficient for specific classes of logarithmic signatures. Furthermore, we develop a generic factorization algo-

rithm, which not only works with logarithmic signatures but all block sequences (and the run-time depends on the structure of the input), and for which the efficiency can be shown for some large classes of logarithmic signatures.

Moreover, we analyze logarithmic signatures of dihedral groups, and present efficient factorization algorithms for specific types of logarithmic signatures.

These results are generalized and extended: we analyze the generalized quaternion group and wreath products.

In the previous investigations we used a specific representation of the group. In another part of the dissertation we analyze in which cases one can give an efficient algorithm for converting elements of an arbitrarily represented group (black box group) with known structure to the representation used in the previous chapters.

Finally, we present our program, in which a cryptosystem (based on a generalized MST_1) and the various generation and factorization algorithms developed in this work have been implemented.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Public-Key Cryptography and Group Factorizations

In a public-key cryptosystem (also called asymmetric cryptosystem), each party has two keys: a private key and a public key. The public key is available to everyone and can be used to encrypt data or to verify a digital signature. The corresponding private key is known only to the owner of the key pair and can be used to decrypt data that has been encrypted using the public key or to create a digital signature.

As the public key P is available to everyone, the mapping E_P that encrypts some data using P must be an injective trap-door function, i.e. an injective one-way function that can be inverted efficiently only when knowing a private key corresponding to P. If it would not be injective, nobody could decrypt the data unambiguously; if it would not be one-way, everybody could decrypt the data using P; and if the private key would not allow efficient inversion, we would have simply a one-way function, not a cryptosystem for data encryption.

Although the public and the private key are related, in a secure public-key cryptosystem it is computationally infeasible to derive a private key from a public one. In contrast, in a symmetric cryptosystem, the two keys are the same or can easily be derived from each other. For a symmetric cryptosystem, a secret key must be exchanged over a secure channel prior to using the cryptosystem; this is not required in a public-key cryptosystem.

Public-key cryptosystems are fundamental to secure communication over insecure networks. Such cryptosystems can be found in many well-known cryptographic protocols, like Hypertext Transfer Protocol Secure (HTTPS) based on Secure Sockets Layer (SSL) or its successor Transport Layer Security (TLS) for secure communication with web servers, Secure/Multipurpose Internet Mail Extensions (S/MIME) and OpenPGP/PGP for e-mail privacy / authentication / integrity / non-repudiation, Secure Shell (SSH) for remote command execution, Secure Copy (SCP) for file transfer, and many others.

Almost all public-key cryptosystems being used today are based on commutative algebraic structures. The two most commonly used public-key cryptosystems are RSA and ElGamal. RSA relies on the assumption that factoring a number into its prime factors is hard for large numbers (and the one-way function is modular exponentiation). ElGamal relies on the hardness of the discrete logarithm problem (DLP) in a cyclic group; it is usually realized either over the cyclic multiplicative group of a field $\mathbb{Z}_p$ (with $p \in \mathbb{P}$) or over the cyclic group generated by a base point on an elliptic curve over a finite field. Lattice-based cryptosystems rely on the hardness of problems (e.g. shortest vector

or closest vector) on lattices over a finite field.

The underlying structures of all these cryptosystems are commutative. Up to now it is unknown whether the properties of commutative structures are a security issue in general. In any case, it is wise to develop and analyze other public-key cryptosystems, especially ones based on non-commutative structures.

One approach for a public-key cryptosystem is based on logarithmic signatures of arbitrary (also including non-commutative, i.e. non-abelian) finite groups. For a finite group G, let α be an ordered set of subsets (blocks) of G. α is called a *logarithmic signature*, if every element of G can be written uniquely as a product of one element from each subset of α (in the given order).

The idea for a one-way function now is the following: choosing one element from each subset of α and computing the product is possible efficiently, but the inversion (i.e. finding the factorization of a group element as a product of one element from each subset of α) may be hard.

If computing factorizations with respect to α is possible in polynomial time for all group elements, α is called *tame*. If no probabilistic polynomial-time algorithm can output the factorization of a randomly chosen group element with a non-negligible probability, α is called *wild*.

One objective is to construct wild logarithmic signatures that additionally allow computing factorizations efficiently when having some extra information (the private key). With this, we have a trap-door function for an asymmetric cryptosystem.

Another objective is to show that certain logarithmic signatures are tame.

1.2. Existing Work

Cryptosystems. S. S. Magliveras [Mag86] has described a symmetric cryptosystem called Permutation Group Mappings (PGM) that uses logarithmic signatures. In PGM, a secret key consists of a pair of tame logarithmic signatures.

S. S. Magliveras, D. R. Stinson and T. van Trung [Mag02b] developed public-key cryptosystems based on logarithmic signatures and $[s, r]$-meshes[1], called MST_1 and MST_2.

Another public-key cryptosystem called MST_3 has been presented by W. Lempken, T. van Trung, S. S. Magliveras and W. Wei [Lem09]. MST_3 is based on the difficulty of factoring group elements with respect to a randomly generated cover[2]. MST_3 requires the group G to have several specific properties (like a large center $\mathcal{Z}$ such that G does not split over $\mathcal{Z}$), and Suzuki 2-groups were proposed explicitly.

A pseudo-random number generator called MST_g has been introduced by P. Svaba in [Sva11] and by P. Marquardt, P. Svaba and T. van Trung in [Mar12]. MST_g is based

[1] An $[s, r]$-mesh is a sequence of s blocks (each containing r group elements) where each group element is generated at least once and the distribution of the generated elements is approximately uniform; see Section 3.1.

[2] A cover is a block sequence that generates every group element at least once.

on random covers. It can be highly efficient, and evidence of excellent statistical and cryptographic properties has been shown.

We give an overview on these cryptosystems in Chapter 4.

PGM. S. S. Magliveras and N. D. Memon [Mag92] discussed various algebraic properties of PGM. Especially, they have shown that the set of all non-trivial transversal[3] logarithmic signature mappings almost always generates the whole symmetric group (by composition). Furthermore, a computationally infeasible chosen-plaintext attack is described.

MST_1. In [Mag02b], it has been shown that transversal logarithmic signatures for permutation groups are tame.

M. I. González Vasco and R. Steinwandt [Vas02] have presented various obstacles for realizing MST_1 and MST_2. This includes a demonstration that not all logarithmic signatures in the class $\mathcal{TNT}(G)$ (the set of all logarithmic signatures for G where no block is a coset of a subgroup of G) are wild. Furthermore, they gave an example of a partial inversion attack.

In [Vas04], M. I. González Vasco, D. Hofheinz, C. Martínez and R. Steinwandt primarily discussed other public-key cryptosystems, but the paper also contains some observations on logarithmic signatures relevant to MST_1. Most importantly, all blocks being anticlosed[4] does not imply being far away from transversal.

J.-M. Bohli, R. Steinwandt, M. I. González Vasco and C. Martínez [Boh05] gave examples of tame totally non-transversal logarithmic signatures for symmetric and alternating groups. Moreover, they presented a method that allows the decision whether a logarithmic signature for a permutation group is exact transversal or not (for this, a binary matrix indicating subgroup generation is constructed and a staircase sequence of ones is searched). This method can also be used in some cases to detect which blocks need to be fused in order to transform a totally non-transversal logarithmic signature into an equivalent transversal one.

A result by S. S. Magliveras and N. D. Memon [Mag92] stating that the set of all non-trivial transversal logarithmic signature mappings almost always generates the whole symmetric group (by composition) is generalized in [Car06]. A. Caranti and F. D. Volta were able to reduce the restrictions on the group: when the group is non-trivial and not cyclic of order p or p^2 for $p \in \mathbb{P}$, the mappings induced by all exact transversal logarithmic signatures generate the whole symmetric group.

In [Bla09], S. R. Blackburn, C. Cid and C. Mullan mostly discussed the realization and the security of MST_3, but the paper also contains an important result that applies to MST_1, too: amalgamated[5] transversal logarithmic signatures for $\mathbb{Z}_2^n$ are tame. The

[3] In a transversal logarithmic signature, each block consists of a full set of coset representatives for a subgroup given by the product of certain other blocks; details can be found in Section 3.4.3.

[4] $S \subseteq G$ is called *anticlosed*, if $st \notin S \setminus \{\mathrm{id}\}$ for all $s, t \in S \setminus \{\mathrm{id}\}$.

[5] A logarithmic signature constructed from an exact transversal logarithmic signature by permuting blocks and elements, translating and fusing blocks; see Section 6.3.

idea for proving this is to recursively move into appropriate factor groups (which can be determined efficiently).

In [Nus11], A. Nuss developed a new security definition for factorization problems (in previous papers, group factorizations may be either tame or wild, i.e. element factorizations can be computed efficiently or not; the new security definition instead is similar to the definition of one-way functions). An efficient factorization algorithm is presented for all logarithmic signatures of cyclic p-groups (which is based on the existence of a periodic[6] block on each recursion level of the algorithm). Another efficient factorization algorithm is presented for all logarithmic signatures of $\mathbb{Z}_2^n$ with block sizes less or equal to 4 (which is based on the existence of a subgroup block on each recursion level of the algorithm). For logarithmic signatures of $\mathbb{Z}_p^n$, an efficient factorization algorithm is presented for the case when a Rédei block[7] exists on each recursion level.

Motivated by the previous attacks, one objective is to find construction methods for aperiodic[8] logarithmic signatures. B. Baumeister and J.-H. de Wiljes [Bau12] introduced a method that generates logarithmic signatures by decomposing and reuniting specific subgroups and transversals.

Building upon this, R. Staszewski and T. van Trung [Sta13] defined strongly aperiodic logarithmic signatures and presented constructions for abelian p-groups (based on the method in [Bau12]).

P. Svaba, T. van Trung and P. Wolf [Sva13] presented factorization algorithms for fused transversal logarithmic signatures of abelian groups (without any restrictions on their representation).

MST$_2$. In [Mag02b], it has been shown that the problem of computing factorizations with respect to covers is at least as hard as the discrete logarithm problem (DLP).

P. Svaba and T. van Trung [Sva07] have shown that generating random covers for large groups is possible efficiently.

MST$_3$. In [Lem09], two computationally infeasible attacks and the space and time complexity of computing with Suzuki 2-groups are discussed. The first attack tries to extract information about the private key from the public key, and the second attack is a chosen-plaintext attack against the private key.

M. I. González Vasco, A. L. Pérez del Pozo and P. Taborda Duarte [Vas10] discussed the security of MST$_3$. Especially, they have shown that the hardness of factoring group elements with respect to random covers for a subset of the group is crucial for the security of MST$_3$ (in the original proposal [Lem09] it was claimed that this is not required in all cases).

[6] A block A (containing elements of the group G) is called *periodic*, if there exists a $g \in G \setminus \{\mathrm{id}\}$ with $Ag = A$ or $gA = A$.

[7] A block A_i of a logarithmic signature $(A_1, \ldots, A_s)$ for a group G is called a *Rédei block*, if $\left\langle \bigcup_{j \neq i} A_j \right\rangle \neq G$.

[8] A logarithmic signature is called *aperiodic*, if it does not contain a periodic block.

P. Svaba and T. van Trung [Sva10] discussed MST_3 in detail. They analyzed MST_3 with Suzuki 2-groups for various classes of group factorizations, present a matrix-permutation attack, and described a concrete implementation (with run-time and space analysis).

Length. For all groups that we consider in this work, logarithmic signatures with minimal length[9] exist. Anyway, we would like to note that there is research on the existence of minimal length logarithmic signatures for various groups, e.g. [Vas03], [Hol04] and [Lem05].

1.3. Organization and Contributions

Our objective is to analyze the realizability and security of MST_1 for various groups and logarithmic signature types.

This work is organized as follows.

1. The introductory Chapter 1 contains a general outline of public-key cryptography and group factorizations, a summary of existing research on the subject, this section about the organization of our work and our contributions, and acknowledgments.

2. Chapter 2 introduces various topics and well-known knowledge required for the subsequent parts of this work. This includes basic group theory, linear representations and characters, definitions and properties of groups, set properties, computational efficiency, one-way functions and cryptographically secure pseudo-random number generators.

   Furthermore, the pseudo-code language used in this work is defined in this chapter.

   Our contributions. We define and analyze antiperiodicity of group subsets.

3. Chapter 3 starts by defining group factorizations and factorization mappings. We define what "tame" and "wild" mean precisely, and we specify classes of block sequences.

4. In Chapter 4, we give an overview on existing cryptographic primitives based on group factorizations.

5. In Chapter 5, various transformations on block sequences are presented. Furthermore, we analyze the irreducibility of blocks and its connection to linear representations and characters.

   Transformations and irreducibility conditions are interesting, because they can be used in block sequence generation and factorization algorithms.

[9] The sum of the block sizes shall be minimal; see Remark 3.5.

Our contributions. Our first main contribution in this chapter is a rigorous analysis of transformations on block sequences. We analyze their effect on factorization mappings, define subclasses of transformations (e.g. factorization-permuting block shuffles, which are interesting for block sequence generation algorithms that include block shuffle transformations). New, interesting types of block sequence normalizations (which we call g- and (i, g)-normalizations) are presented. Block substitutions are introduced, and we have a look at their efficiency and a possible implementation. We analyze compositions of transformations in detail (e.g. which compositions form a group); this is especially important for designing block sequence generation algorithms based on iterated transformations. In light of this, we emphasize one special combination (translation, element shuffle and normalization), which we call a TSN transformation and which plays a significant role in our logarithmic signature generation algorithm presented in Chapter 6.

Another contribution in this chapter is an analysis of irreducibility of blocks, mainly based on group homomorphisms. Linear representations and characters are useful for studying factorizations of abelian groups; we contribute an analysis that shows to what extent some results for abelian groups can be generalized for non-abelian groups.

6. In Chapter 6, we review existing algorithms for generating logarithmic signatures, and present a new generation algorithm.

   Our contributions. We design a new algorithm called LS-Gen that generates logarithmic signatures, which may possibly be wild when the generation procedure is kept secret. In contrast to most other existing generation algorithms, LS-Gen also works with non-abelian groups. The algorithm is based on iteratively applying transformations on an initial logarithmic signature. We discuss the algorithm's security and the internal interactions of the components of the algorithm. Furthermore, we show that for abelian groups LS-Gen typically generates a larger set of logarithmic signatures than the amalgamated ones (generated by the most commonly used algorithm in literature).

7. In Chapter 7, we have a look at factorization approaches for arbitrary groups, in a rather abstract way.

   Our contributions. We show that regarding only canonical block sequences is not a real restriction. We consider using homomorphisms, both for logarithmic signatures and multiple factorizations. We furthermore consider moving into factor groups via normal subgroup blocks. A demonstration with direct products shows that a simple concatenation approach does not work.

8. In Chapter 8, we regard factorizations of abelian groups G, represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$.

Our contributions. Our contributions in this chapter start with a detailed analysis of the structure of logarithmic signatures $\alpha \in \Lambda(\mathbb{Z}_{2^n})$ of type $t(\alpha) = (2, 2, \ldots, 2)$. This leads to an efficient factorization algorithm for logarithmic signatures $\alpha \in \Lambda(\mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^k)$ of type $t(\alpha) = (2, 2, \ldots, 2)$.

Based on a theorem of Rédei, we show that for all abelian groups every logarithmic signature $\alpha = (A_1, A_2, \ldots, A_n) \in \Lambda(G)$ with $|A_i| \in \mathbb{P}$ for all $1 \leq i \leq n$ is tame. This especially includes a detailed elaboration on how to efficiently compute in factor groups.

Building upon the previous result, we design more factorization algorithms. First, we drop the requirement that blocks need to be of prime size and point out that the previous algorithm still works fine if it finds a subgroup on each recursion level. We further generalize this by showing that finding a periodic block on each recursion level is sufficient.

We then design a generic factorization algorithm, which is defined for all block sequences (e.g. the input block sequence does not necessarily have to contain a periodic block). We justify why this algorithm is efficient when periodic blocks exist on each recursion level (i.e. the generic algorithm supersedes all previous factorization algorithms), argue why it is efficient even for aperiodic logarithmic signatures generated by an algorithm in Chapter 6, and point out why it even may be efficient for logarithmic signatures generated by LS-Gen in practice.

We have a look at other factorization approaches, e.g. via recursively moving into statically or dynamically chosen factor groups, and demonstrate obstacles that can occur.

Using our algorithms, we show that various classes of logarithmic signatures are tame now: amalgamated transversal, aperiodically decomposed and reunited for $\mathbb{Z}_2^n$ from [Bau12], and strongly aperiodic constructions from [Sta13].

We provide a list of logarithmic signatures (of specific types and for specific groups) that can be proved to be tame.

We describe how a factorization problem can be modeled as an integer linear programming (ILP) problem. It turns out that none of the ILP solvers that we tested are able to find factorizations efficiently; we present run-time examples.

Finally, we count logarithmic signatures for some specific abelian groups and logarithmic signature types.

9. In Chapter 9, we study factorizations of dihedral groups. The dihedral group D_{2n} can be described as the symmetry group of a regular polygon having n sides. Dihedral groups in some sense are the most simple non-abelian groups.

   Our contributions. We first present several interesting block substitution transformations (both unconditional and conditional ones) in dihedral groups, which are later used both in our generation and factorization algorithms. We especially analyze size-permutable blocks in detail.

Subsequently, we design factorization algorithms (some efficient, some not) for various special cases, including the case when all blocks of $\alpha \in \Lambda(D_{2n})$ except one of size 2 contain rotations only (the run-time of our algorithm depends on the run-time of another factorization algorithm for a $\mathbb{Z}_n$ logarithmic signature), $D_{2 \cdot p^n}$ (our run-time depends on the structure of the logarithmic signature), the special case $\alpha \in \Lambda(D_{2 \cdot 2^n})$ of type $t(\alpha) = (2, 2, \ldots, 2)$ (the algorithm is always efficient, i.e. α is tame; note that α has minimal length). We slightly generalize the last case by showing that any logarithmic signature $\alpha \in \Lambda(D_{2 \cdot 2^n} \times \mathbb{Z}_2^k)$ of type $t(\alpha) = (2, 2, \ldots, 2)$ is tame.

Furthermore, we design and analyze an algorithm to generate logarithmic signatures for D_{2n} with n odd.

Similar to the previous chapter, we close this chapter by counting logarithmic signatures for some specific dihedral groups and logarithmic signature types.

10. In Chapter 10, we regard factorizations of other groups.

    Our contributions. First we regard generalized quaternion groups, and show that every $\alpha \in \Lambda(Q_{4 \cdot 2^n})$ of type $t(\alpha) = (2, 2, \ldots, 2)$ is tame. Building upon this, we prove that every $\alpha \in \Lambda(G)$ of type $t(\alpha) = (2, 2, \ldots, 2)$ is tame for all non-abelian groups of order $|G| = 2^{n+1}$ (with $n \geq 3$) with a cyclic $N \trianglelefteq G$ of order $|N| = 2^n$. For non-abelian groups G of order $|G| = p^{n+1}$ (with $p \in \mathbb{P}_{\geq 3}$ and $n \geq 2$) with a cyclic $N \trianglelefteq G$ of order $|N| = p^n$, $\alpha \in \Lambda(G)$ of type $t(\alpha) = (p, p, \ldots, p)$, we present a reduction to the factorization problem with respect to a p-factorization of $\mathbb{Z}_{p^{n-1}} \oplus \mathbb{Z}_p$, and prove the tameness in a special case.

    We then generalize the previous result using homomorphisms with small kernels. As a corollary we show that every $\alpha \in \Lambda(G)$ of type $t(\alpha) = (2, 2, \ldots, 2)$ is tame when G is an extra special 2-group.

    Finally, we regard wreath products. We present a few factorization approaches, including an orbit-based factor group descending method.

11. In Chapter 11, we regard black box groups. In all previous chapters we assumed that the groups were given in a specific representation (e.g. in Chapter 8, we assumed the abelian group G to be represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$), and the algorithms in these chapters usually depend on these specific representations. In Chapter 11, we assume that a group is given as a black box (i.e. any arbitrary representation), supporting only a few basic group operations. Given such a black box group, our goal is to map elements to the representations that we require in the previous chapters. If we succeed, we have shown that the results from the previous chapters actually hold for arbitrary representations of a group with the specific structure.

    Our contributions. Our first result is that cyclic black box groups indeed can be mapped to our usual representation $\mathbb{Z}_n$ (integers mod n). For elementary abelian p-groups, we show a few results for the case when an efficient linear

dependence test is available. A few weaker results are presented for the general case of an abelian group. For dihedral groups, we show that the mapping is possible efficiently. Furthermore, we show that every $\alpha \in \Lambda(G)$ of type $t(\alpha) = (2, 2, \ldots, 2)$ is tame when G is a non-abelian group (given as black box group) of order $|G| = 2^{n+1}$ having a cyclic normal subgroup $N \trianglelefteq G$ of order $|N| = 2^n$.

12. **Our contributions.** We have implemented most of the algorithms mentioned in this work. Chapter 12 documents our program LogSig, which is written in C# and runs on Windows, Linux and Mac OS X. The program supports encrypting/decrypting files using a cryptosystem based on a generalized MST_1, where the logarithmic signatures are generated using our LS-Gen algorithm mentioned before. Furthermore, factorization algorithms are implemented, especially including our powerful generic factorization algorithm.

13. In Chapter 13, we point out various possibilities for further research.

1.4. Acknowledgments

I would especially like to thank my doctoral advisor Prof. Dr. Peter Hauck for his excellent guidance and great support, both for this dissertation and all my other activities at University (tutorial groups, etc.). I cannot imagine a better mentor.

Thanks a lot to Prof. Dr. Klaus Reinhardt for being the second referee for this work.

Many thanks to Prof. Dr. Klaus-Jörn Lange and Jun. Doz. Dr. Britta Dorn for being examiners in the oral examination.

Additionally, thanks to all other members of the Discrete Mathematics group (especially including Claudia Schmidt, Juliane Bertram and Stephanie Reifferscheid); it was a pleasure to work with you. Furthermore, thanks to Renate Hallmayer, who always had time for my administrative matters.

Thanks to Bernd Borchert for interesting discussions about various cryptographic topics.

Thanks to the University of Tübingen for supporting this work with a dissertation grant (Promotionsstipendium nach dem Landesgraduiertenförderungsgesetz).

Last but not least I would like to thank my parents and my grandmother for their support during my whole time at University. This dissertation is dedicated to you.

2. Preliminaries

Chapter 2 introduces various topics and well-known knowledge required for the subsequent parts of this work. This includes basic group theory, linear representations and characters, definitions and properties of groups, set properties, computational efficiency, one-way functions and cryptographically secure pseudo-random number generators.

Furthermore, the pseudo-code language used in this work is defined in this chapter.

Our contributions. We define and analyze antiperiodicity of group subsets.

2.1. Group Theory

Let G be a set and $\cdot : G \times G \to G$ a binary operation on G. $(G, \cdot)$ is called a *group*, if:

- $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in G$ (*associativity*).
- There exists an element $e \in G$, such that $e \cdot g = g \cdot e = g$ for all $g \in G$ (e is called *identity element*; it is unique, thus we speak of *the* identity element).
- For every element $g \in G$ there exists an element $g^{-1} \in G$ with $g \cdot g^{-1} = g^{-1} \cdot g = e$ (g^{-1} is called the *inverse element* of g).

G is called the *underlying set* of the group $(G, \cdot)$. Usually we just write G instead of $(G, \cdot)$, when the operation is clear. Also, the operation is sometimes omitted; "gh" is short for "$g \cdot h$".

The identity element e will usually be denoted by id (when the operation symbol is "$\cdot$") or 0 (when the operation symbol is "+").

The *order* $|G|$ of the group G is the number of elements in the set G.

All groups in this work are by default assumed to be finite, i.e. $|G|$ is finite.

G is called a *p-group*, if $|G| = p^m$ for some $p \in \mathbb{P}$ and $m \in \mathbb{N}_0$.

For an element $g \in G$ define $\langle g \rangle := \{g^n \mid n \in \mathbb{Z}\}$. The *order* of g is $\mathrm{ord}(g) := |\langle g \rangle|$. If the order of an element g is finite (which is always the case in a finite group), it is the minimal $n > 0$ for which $g^n = \mathrm{id}$.

A subset $\emptyset \neq U \subseteq G$ is a *subgroup* $(U, \cdot)$ of $(G, \cdot)$, if for all $g, h \in U$ the product $g \cdot h$ and the inverse g^{-1} are in U. When U is a subgroup of a group G, this is denoted by $U \leq G$.

For a subset $S \subseteq G$, $\langle S \rangle$ denotes the smallest subgroup of G containing S (such a subgroup exists; it is the intersection of all subgroups containing S).

Given an element $g \in G$ and a subgroup $U \leq G$, the *left coset* of U containing g is $gU := \{g \cdot u \mid u \in U\}$. Respectively, the *right coset* of U containing g is $Ug := \{u \cdot g \mid u \in U\}$. The cosets of U form a partition of G (i.e. the union of all cosets covers G and each two cosets are either equal or their intersection is empty).

A subgroup $N \leq G$ is called a *normal subgroup*, if $gN = Ng$ for all $g \in G$. When N is a normal subgroup of a group G, this is denoted by $N \trianglelefteq G$.

When $N \trianglelefteq G$ is a normal subgroup of G, then $G/N := (\{gN \mid g \in G\}, *)$ with the binary operation $* : G/N \times G/N \to G/N : (gN, hN) \mapsto ghN$ is also a group, called *factor group*.

$Z(G) := \{z \in G \mid zg = gz \text{ for all } g \in G\}$ is the *center* of G. The center of a group is always a normal subgroup.

$C_G(S) := \{g \in G \mid sg = gs \text{ for all } s \in S\}$ is the *centralizer* of the subset $S \subseteq G$. We have $C_G(S) \leq G$.

$\mathrm{Cl}(g) := \{hgh^{-1} \mid h \in G\}$ denotes the *conjugacy class* of a $g \in G$. The conjugacy classes form a partition of the elements of G. For a subset or multiset $A \subseteq G$, we write $\mathrm{Cl}(A) := [\mathrm{Cl}(g) \mid g \in A]$ for the multiset of conjugacy classes of the elements in A.

A group G is called *abelian*, if $g + h = h + g$ holds for all $g, h \in G$. In this work we usually write abelian groups additively (i.e. the operation symbol is "+") and non-abelian groups multiplicatively (i.e. the operation symbol is "$\cdot$").

If there exists a $g \in G$ such that $\langle g \rangle = G$, then G is called *cyclic*.

An abelian p-group G is called *elementary abelian*, if $\mathrm{ord}(g) = p$ holds for all $g \in G \setminus \{0\}$.

Let G be a p-group. If $|Z(G)| = p$ and $G/Z(G)$ is a non-trivial elementary abelian p-group, then G is called *extra special*.

For $A, B \subseteq G$, $A \cdot B$ (or short AB) denotes the product $A \cdot B := \{a \cdot b \mid a \in A, b \in B\}$. If G is written additively, analogously $A + B := \{a + b \mid a \in A, b \in B\}$ denotes the sum.

For two groups $(G, \cdot)$ and $(H, *)$, $G \times H$ denotes the *direct product*, i.e. the elements of $G \times H$ are ordered pairs (g, h) with $g \in G$ and $h \in H$, which together with the following binary operation form a new group:

$$\bullet : (G \times H) \times (G \times H) \to G \times H : ((g_1, h_1), (g_2, h_2)) \mapsto (g_1 \cdot g_2, h_1 * h_2).$$

If G and H are abelian, $G \oplus H$ denotes the *direct sum*, which is just the direct product written additively.

We write G^n for the direct product/sum of n copies of G, i.e.

$$G^n = \underbrace{G \times G \times \ldots \times G}_{n \text{ times}}.$$

Let $(G, \cdot)$ and $(H, *)$ be groups. A *group homomorphism* is a function $\varphi : G \to H$ with

$$\varphi(a \cdot b) = \varphi(a) * \varphi(b)$$

for all $a, b \in G$.

The *kernel* $\ker(\varphi) := \{g \in G \mid \varphi(g) = \mathrm{id}_H\}$ is a normal subgroup of G. For every $N \trianglelefteq G$ there exists a homomorphism that has N as kernel. The image $\mathrm{im}(\varphi) := \varphi(G) = \{\varphi(g) \mid g \in G\}$ is a subgroup of H.

Two groups are called *isomorphic*, if there exists a bijective homomorphism between them; this is denoted using the symbol "$\cong$". For a group homomorphism $\varphi : G \to H$, we have $\mathrm{im}(\varphi) \cong G/\ker(\varphi)$.

A bijective homomorphism is called an *isomorphism*. An isomorphism from a group to itself is called an *automorphism*. The set of all automorphisms on a group G is denoted by $\mathrm{Aut}(G)$. With composition as binary operation, $\mathrm{Aut}(G)$ is a group.

The group of *inner automorphisms* is $\mathrm{Inn}(G) := \{\varphi : G \to G : x \mapsto g^{-1}xg \mid g \in G\}$.

The group of *central automorphisms* is $\mathrm{Aut}_c(G) := \{\varphi \in \mathrm{Aut}(G) \mid g^{-1}\varphi(g) \in Z(G) \text{ for all } g \in G\}$; we have $\mathrm{Aut}_c(G) = C_{\mathrm{Aut}(G)}(\mathrm{Inn}(G))$.

Let N and H be groups and $\varphi : H \to \mathrm{Aut}(N)$ a group homomorphism. The set $N \times H$ together with the binary operation

$$((n_1, h_1), (n_2, h_2)) \mapsto (n_1 \cdot \varphi(h_1)(n_2), h_1 \cdot h_2)$$

forms a group and is called the *semidirect product* $N \rtimes_\varphi H$ of N and H with respect to φ.

Let G be an abelian group. According to the fundamental theorem of finitely generated abelian groups [Hup67], G is isomorphic to $\mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$ (where $\mathbb{Z}_n$ is the cyclic group of order n). In this representation, $\mathbb{Z}_{p_1^{k_1}}, \ldots, \mathbb{Z}_{p_m^{k_m}}$ are called the *components* of G. The *type* of G is $t(G) := (p_1^{k_1}, \ldots, p_m^{k_m})$.

$(K, +, \cdot)$ (a set K with two binary operations $+$ and $\cdot$) is called a *field*, if $(K, +)$ is an abelian group (with identity element 0), $(K \setminus \{0\}, \cdot)$ is an abelian group (called *multiplicative group*) and $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ holds for all $a, b, c \in K$ (*distributivity*).

2.2. Linear Representations and Characters

Let G be a group, K a field and $n \in \mathbb{N}$. $\mathrm{GL}(n, K)$ denotes the general linear group of degree n over K, i.e. $\mathrm{GL}(n, K) := \{M \in \mathcal{M}_n(K) \mid M \text{ is invertible}\}$.

A group homomorphism $\rho : G \to \mathrm{GL}(n, K)$ is called a *K-representation* of G.

If $\rho(g) = \mathrm{id}$ for all $g \in G$, ρ is called the *trivial representation*. ρ is called *faithful*, if and only if $\rho(G) \cong G$ (or equivalently $\ker(\rho) = \{\mathrm{id}\}$).

ρ defines an action on the vector space K^n. If ρ is non-trivial and there exists no non-trivial proper invariant subspace, ρ is called *irreducible*.

In the rest of this section, we assume $K = \mathbb{C}$.

Let G be a group and $\rho : G \to \mathrm{GL}(n, \mathbb{C})$ a $\mathbb{C}$-representation of finite dimension $n < \infty$. The *character* with respect to ρ is defined as

$$\chi_\rho : G \to \mathbb{C} : g \mapsto \mathrm{tr}(\rho(g)).$$

χ_ρ is a class function on G, i.e. it is constant on each conjugacy class: $\chi_\rho(hgh^{-1}) = \chi_\rho(g)$ for all $g, h \in G$.

A *character table* of a group G is a table in which each column corresponds to a conjugacy class c_j of G and each row to the character χ_{ρ_i} of an irreducible representation ρ_i of G. In the entries, the image $\chi_{\rho_i}(c_j)$ is listed.

From representation/character theory we know:

- There are exactly as many conjugacy classes as irreducible representations, i.e. the character table is square.
- Row orthogonality. $(f|h) := \frac{1}{|G|} \sum_{g \in G} f(g)\overline{h(g)}$ is an hermitian inner product, and for the characters $\chi_1, \chi_2, \ldots$ in the character table we have $(\chi_i|\chi_j) = \delta_{i,j}$, i.e. the χ_i are orthogonal.
- Column orthogonality. $\frac{1}{|C_G(g_i)|} \sum_{\chi_k} \chi_k(g_i)\overline{\chi_k(g_j)} = \delta_{i,j}$.

If there are d irreducible representations/characters, it can be proven that the following equation holds:

$$\sum_{i=1}^{d} \chi_i(\mathrm{id})^2 = |G|.$$

If G is abelian, all irreducible representations of G are of degree 1.

2.3. Groups

2.3.1. Permutation Group

Let X be a set. The group of all permutations acting on X is called the *symmetric group* on X, and is denoted by $\mathrm{Sym}(X)$. If X is actually an algebraic structure, $\mathrm{Sym}(X)$ acts on the underlying set of elements.

If $X = \{1, 2, \ldots, n\}$, we also write $\mathrm{Sym}(n)$ for $\mathrm{Sym}(X)$.

For example, we have $\mathrm{Sym}(6) = \mathrm{Sym}(\{1, 2, 3, 4, 5, 6\}) \cong \mathrm{Sym}(\mathbb{Z}_6)$.

We write $\mathrm{sgn}(\pi)$ for the sign of the permutation π (i.e. $\mathrm{sgn}(\pi) := (-1)^{N(\pi)}$ with $N(\pi)$ the number of inversions in π). If $\mathrm{sgn}(\pi) = 1$, π is called *even*; otherwise (i.e. when $\mathrm{sgn}(\pi) = -1$), π is called *odd*. The function $\mathrm{sgn} : \mathrm{Sym}(X) \to \{1, -1\}$ is a group homomorphism ($\{1, -1\}$ and multiplication as binary operation form a group).

The kernel of sgn is called the *alternating group* $\mathrm{Alt}(X)$. Analogously to the notation for symmetric groups, in the case $X = \{1, 2, \ldots, n\}$ we also write $\mathrm{Alt}(n)$ for $\mathrm{Alt}(X)$.

Let G be a permutation group. We define two functions to count the number of even and odd permutations in a set or multiset of elements:

$$\mathsf{c}_e : \mathcal{P}(G) \to \mathbb{N}_0 : A \mapsto |\{\pi \in A \mid \mathrm{sgn}(\pi) = 1\}|,$$

$$\mathsf{c}_o : \mathcal{P}(G) \to \mathbb{N}_0 : A \mapsto |\{\pi \in A \mid \operatorname{sgn}(\pi) = -1\}|\,.$$

For a set or multiset A, we have $|A| = \mathsf{c}_e(A) + \mathsf{c}_o(A)$.

2.3.2. Cyclic Group

$\mathbb{Z}_n := (\mathbb{Z}/n\mathbb{Z}, +)$ denotes the cyclic group of order n. We use the numbers $\{0, 1, \ldots, n-1\}$ as canonical representatives and expect the "+" operator to automatically perform a modulo computation:

$$+ : \mathbb{Z}_n^2 \to \mathbb{Z}_n : (x, y) \mapsto (x + y) \bmod n.$$

We write $\mathbb{Z}_n^k$ (not $k\mathbb{Z}_n$) for the direct sum of k copies of $\mathbb{Z}_n$, i.e.

$$\mathbb{Z}_n^k = \underbrace{\mathbb{Z}_n \oplus \mathbb{Z}_n \oplus \ldots \oplus \mathbb{Z}_n}_{k \text{ times}}.$$

2.3.3. Dihedral Group

2.3.3.1. Definition and Fundamental Properties

Let $n \in \mathbb{N}$. The dihedral group D_{2n} can be described as the symmetry group of a regular polygon having n sides.

In a Cartesian coordinate system in two dimensions, let the center of the polygon be the origin, and let one vertex of the polygon lie on the positive half of the x-axis (i.e. one vertex has the coordinates $(r, 0)$ with $0 < r \in \mathbb{R}$).

We write σ for a counterclockwise rotation by the angle $\frac{2\pi}{n}$, and τ for a reflection across the x-axis. With this, each element of D_{2n} can be written uniquely in the form $\sigma^k\tau^c$ with $k \in \{0, 1, \ldots, n-1\}$ and $c \in \{0, 1\}$.

An element $\sigma^k\tau^c \in D_{2n}$ is called a *rotation*, if $c = 0$. If $c = 1$, the element is called a *reflection*.

The order of D_{2n} is $2n$, and we have $\operatorname{ord}(\sigma) = n$, $\operatorname{ord}(\tau) = 2$. An important equation (for multiplying elements) is

$$\sigma \cdot \tau = \tau \cdot \sigma^{-1} = \tau \cdot \sigma^{n-1}.$$

In presentation notation, D_{2n} is $\langle s, t \mid s^n = t^2 = \mathrm{id},\, t^{-1}st = s^{-1}\rangle$.

The dihedral group is interesting, because it is one of the most simple non-abelian finite groups. For $n > 2$, D_{2n} is non-abelian (because for example we have $\sigma^2\tau \cdot \sigma\tau = \sigma \neq \sigma^{n-1} = \sigma^{-1} = \sigma\tau \cdot \sigma^2\tau$ for $n > 2$). For $n = 2$, $D_{2\cdot 2}$ is isomorphic to the Klein four-group $\mathbb{Z}_2^2$.

Notation. In order to save some space (especially in the logarithmic signature

examples), we sometimes omit the σ. We denote σ^i by i and $\sigma^i\tau$ by $i\tau$.

We define two functions to count the number of rotations and reflections in a set or multiset of elements in $G = D_{2n}$:

$$\mathscr{C}_\sigma : \mathcal{P}(G) \to \mathbb{N}_0 : A \mapsto |\{g \in A \mid g \text{ is a rotation}\}|,$$
$$\mathscr{C}_\tau : \mathcal{P}(G) \to \mathbb{N}_0 : A \mapsto |\{g \in A \mid g \text{ is a reflection}\}|.$$

For a set or multiset A, we have $|A| = \mathscr{C}_\sigma(A) + \mathscr{C}_\tau(A)$.

Lemma 2.1. *Let $G = D_{2n}$ with $n > 2$. Then*

$$\mathrm{Aut}(G) = \{\varphi_{m,l} : G \to G : \sigma^k\tau^b \mapsto \sigma^{mk+bl}\tau^b \mid m \in \mathbb{Z}_n^*, l \in \mathbb{Z}_n\}.$$

Proof. This is well known, but for the convenience of the reader we provide a proof.

As $m \in \mathbb{Z}_n^*$ (i.e. $\gcd(m, n) = 1$), $\varphi_{m,l}$ is bijective. Let $g = \sigma^u\tau^v, h = \sigma^x\tau^y \in G$ with $u, x \in \mathbb{Z}_n$ and $v, y \in \{0, 1\}$. $\varphi_{m,l}$ is a homomorphism:

- If $v = 0$:

$$\begin{aligned}\varphi_{m,l}(gh) &= \varphi_{m,l}(\sigma^u\sigma^x\tau^y) = \varphi_{m,l}(\sigma^{u+x}\tau^y) = \sigma^{m(u+x)+yl}\tau^y = \sigma^{mu}\sigma^{mx+yl}\tau^y \\ &= \varphi_{m,l}(\sigma^u) \cdot \varphi_{m,l}(\sigma^x\tau^y) = \varphi_{m,l}(g) \cdot \varphi_{m,l}(h).\end{aligned}$$

- If $v = 1$ and $y = 0$:

$$\begin{aligned}\varphi_{m,l}(gh) &= \varphi_{m,l}(\sigma^u\tau\sigma^x) = \varphi_{m,l}(\sigma^{u-x}\tau) = \sigma^{m(u-x)+l}\tau = \sigma^{mu+l}\tau\sigma^{mx} \\ &= \varphi_{m,l}(\sigma^u\tau) \cdot \varphi_{m,l}(\sigma^x) = \varphi_{m,l}(g) \cdot \varphi_{m,l}(h).\end{aligned}$$

- If $v = 1$ and $y = 1$:

$$\begin{aligned}\varphi_{m,l}(gh) &= \varphi_{m,l}(\sigma^u\tau\sigma^x\tau) = \varphi_{m,l}(\sigma^{u-x}) = \sigma^{m(u-x)} = \sigma^{mu}\tau\sigma^{mx}\tau \\ &= \sigma^{mu+l}\tau\sigma^{mx+l}\tau = \varphi_{m,l}(\sigma^u\tau) \cdot \varphi_{m,l}(\sigma^x\tau) = \varphi_{m,l}(g) \cdot \varphi_{m,l}(h).\end{aligned}$$

These homomorphisms form a group: we have $\varphi_{1,0} = \mathrm{id}$, $\varphi_{m_2,l_2}(\varphi_{m_1,l_1}(\sigma^k\tau^b)) = \varphi_{m_2,l_2}(\sigma^{m_1k+bl_1}\tau^b) = \sigma^{m_2(m_1k+bl_1)+bl_2}\tau^b = \sigma^{m_2m_1k+b(m_2l_1+l_2)}\tau^b = \varphi_{m_2m_1,m_2l_1+l_2}(\sigma^k\tau^b)$ and $\varphi_{m,l}^{-1} = \varphi_{m^{-1},-l\cdot m^{-1}}$ (as $\varphi_{m^{-1},-l\cdot m^{-1}}(\varphi_{m,l}(\sigma^k\tau^b)) = \varphi_{m^{-1},-l\cdot m^{-1}}(\sigma^{mk+bl}\tau^b) = \sigma^{m^{-1}(mk+bl)+b(-l\cdot m^{-1})}\tau^b = \sigma^k\tau^b$).

Let φ be an automorphism of G. We have $\varphi(\sigma) = \sigma^m$ for an $m \in \mathbb{Z}_n$ (due to $\langle\sigma\rangle$ being characteristic in G); and as φ is invertible, we must have $\gcd(m, n) = 1$, i.e. indeed $m \in \mathbb{Z}_n^*$. Furthermore, $\varphi(\tau) = \sigma^l\tau$ (τ cannot be sent to $\langle\sigma\rangle$). Thus indeed $\varphi = \varphi_{m,l}$ for some $m \in \mathbb{Z}_n^*$ and $l \in \mathbb{Z}_n$. □

2.3.3.2. Conjugacy Classes

Lemma 2.2. *Let $G = D_{2n}$. If n is odd, the conjugacy classes in G are* $\{\mathrm{id}\}$, $\{\sigma, \sigma^{-1}\}$, $\{\sigma^2, \sigma^{-2}\}$, …, $\{\sigma^{\frac{n-1}{2}}, \sigma^{-\frac{n-1}{2}}\}$, $\{\sigma^k\tau \mid 0 \leq k \leq n-1\}$. *If n is even, the conjugacy classes in G are* $\{\mathrm{id}\}$, $\{\sigma, \sigma^{-1}\}$, $\{\sigma^2, \sigma^{-2}\}$, …, $\{\sigma^{\frac{n}{2}-1}, \sigma^{-(\frac{n}{2}-1)}\}$, $\{\sigma^{\frac{n}{2}}\}$, $\{\sigma^{2k}\tau \mid 0 \leq k \leq \frac{n}{2}-1\}$, $\{\sigma^{2k+1}\tau \mid 0 \leq k \leq \frac{n}{2}-1\}$.

Proof. First of all, each rotation σ^k together with its inverse σ^{-k} forms a conjugacy class, because for an arbitrary σ^l we have $\sigma^l\sigma^k\sigma^{-l} = \sigma^k$ and for an arbitrary $\sigma^l\tau$ we have $\sigma^l\tau\sigma^k\sigma^l\tau = \sigma^{-k}$. The cases id and $\sigma^{\frac{n}{2}}$ in the even case are special, because they are self-inverse.

Now the reflections. Let $\sigma^k\tau$ be a reflection and l an arbitrary integer. Then $\sigma^l\sigma^k\tau\sigma^l = \sigma^{k+2l}\tau$ and $\sigma^l\tau\sigma^k\tau\sigma^l\tau = \sigma^{-k+2l}\tau$.

In the case where n is odd, 2 is coprime to n and thus every integer $j \in \{0, 1, \ldots, n-1\}$ can be expressed as $j = k + 2l \bmod n$ with an appropriate l. Consequently, the conjugacy class of $\sigma^k\tau$ is the subset of all reflections.

In the case where n is even, 2 is not coprime to n anymore. If k in $\sigma^k\tau$ is even and l variable, we get all reflections with an even rotation component, because $k+2l \bmod n$ and $-k+2l \bmod n$ are even (for even n). Similarly, if k in $\sigma^k\tau$ is odd and l variable, we get all reflections with an odd rotation component, because $k+2l \bmod n$ and $-k+2l \bmod n$ are odd. □

2.3.3.3. Permutation Representation

The representation of elements of D_{2n} in Section 2.3.3.1 can efficiently be converted to a representation using permutations in $\mathrm{Sym}(n)$.

Permutation representation of D_{2n}. Let $\sigma^k\tau^c \in D_{2n}$ (with $0 \leq k < n$ and $c \in \{0, 1\}$). The corresponding permutation $p = (p_1, p_2, \ldots, p_n) \in \mathrm{Sym}(n)$ (where p_i is the image $p(i)$) can be computed as follows:

- For $i \leftarrow 1$ to n:
  - If $c = 0$: set $p_i \leftarrow ((i + k - 1) \bmod n) + 1$,
    else (i.e. $c = 1$): set $p_i \leftarrow (-(i + k - 1) \bmod n) + 1$.

Symmetry group representation corresponding to a permutation in D_{2n}. Let $p \in \mathrm{Sym}(n)$. To compute the corresponding element $\sigma^k\tau^c$ in D_{2n}:

1. If $p(2) - p(1) \equiv 1 \pmod n$: set $c \leftarrow 0$,
   else: set $c \leftarrow 1$.

2. If $c = 0$: set $k \leftarrow p(1) - 1$,
   else: set $k \leftarrow 1 - p(1) \bmod n$.

Example ($D_{2\cdot 4}$).

Symm. gr. rep.	Image arr.	Cycle rep.	sgn
id	1 2 3 4	(1)(2)(3)(4)	1
σ	2 3 4 1	(1 2 3 4)	-1
σ^2	3 4 1 2	(1 3)(2 4)	1
σ^3	4 1 2 3	(1 4 3 2)	-1
τ	1 4 3 2	(1)(2 4)(3)	-1
$\sigma\tau$	4 3 2 1	(1 4)(2 3)	1
$\sigma^2\tau$	3 2 1 4	(1 3)(2)(4)	-1
$\sigma^3\tau$	2 1 4 3	(1 2)(3 4)	1

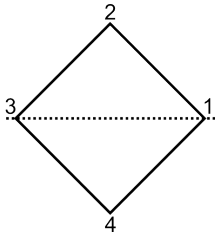

Figure 2.1.: Square

Example ($D_{2\cdot 5}$).

Symm. gr. rep.	Image arr.	Cycle rep.	sgn
id	1 2 3 4 5	(1)(2)(3)(4)(5)	1
σ	2 3 4 5 1	(1 2 3 4 5)	1
σ^2	3 4 5 1 2	(1 3 5 2 4)	1
σ^3	4 5 1 2 3	(1 4 2 5 3)	1
σ^4	5 1 2 3 4	(1 5 4 3 2)	1
τ	1 5 4 3 2	(1)(2 5)(3 4)	1
$\sigma\tau$	5 4 3 2 1	(1 5)(2 4)(3)	1
$\sigma^2\tau$	4 3 2 1 5	(1 4)(2 3)(5)	1
$\sigma^3\tau$	3 2 1 5 4	(1 3)(2)(4 5)	1
$\sigma^4\tau$	2 1 5 4 3	(1 2)(3 5)(4)	1

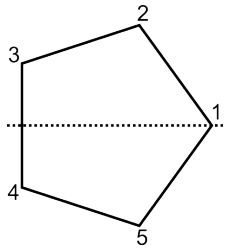

Figure 2.2.: Regular Pentagon

Example ($D_{2\cdot 6}$).

Symm. gr. rep.	Image arr.	Cycle rep.	sgn
id	1 2 3 4 5 6	(1)(2)(3)(4)(5)(6)	1
σ	2 3 4 5 6 1	(1 2 3 4 5 6)	-1
σ^2	3 4 5 6 1 2	(1 3 5)(2 4 6)	1
σ^3	4 5 6 1 2 3	(1 4)(2 5)(3 6)	-1
σ^4	5 6 1 2 3 4	(1 5 3)(2 6 4)	1
σ^5	6 1 2 3 4 5	(1 6 5 4 3 2)	-1
τ	1 6 5 4 3 2	(1)(2 6)(3 5)(4)	1
$\sigma\tau$	6 5 4 3 2 1	(1 6)(2 5)(3 4)	-1
$\sigma^2\tau$	5 4 3 2 1 6	(1 5)(2 4)(3)(6)	1
$\sigma^3\tau$	4 3 2 1 6 5	(1 4)(2 3)(5 6)	-1
$\sigma^4\tau$	3 2 1 6 5 4	(1 3)(2)(4 6)(5)	1
$\sigma^5\tau$	2 1 6 5 4 3	(1 2)(3 6)(4 5)	-1

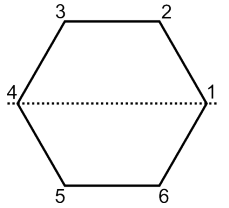

Figure 2.3.: Regular Hexagon

Example ($D_{2\cdot 7}$).

Symm. gr. rep.	Image arr.	Cycle rep.	sgn
id	1 2 3 4 5 6 7	(1)(2)(3)(4)(5)(6)(7)	1
σ	2 3 4 5 6 7 1	(1 2 3 4 5 6 7)	1
σ^2	3 4 5 6 7 1 2	(1 3 5 7 2 4 6)	1
σ^3	4 5 6 7 1 2 3	(1 4 7 3 6 2 5)	1
σ^4	5 6 7 1 2 3 4	(1 5 2 6 3 7 4)	1
σ^5	6 7 1 2 3 4 5	(1 6 4 2 7 5 3)	1
σ^6	7 1 2 3 4 5 6	(1 7 6 5 4 3 2)	1
τ	1 7 6 5 4 3 2	(1)(2 7)(3 6)(4 5)	-1
$\sigma\tau$	7 6 5 4 3 2 1	(1 7)(2 6)(3 5)(4)	-1
$\sigma^2\tau$	6 5 4 3 2 1 7	(1 6)(2 5)(3 4)(7)	-1
$\sigma^3\tau$	5 4 3 2 1 7 6	(1 5)(2 4)(3)(6 7)	-1
$\sigma^4\tau$	4 3 2 1 7 6 5	(1 4)(2 3)(5 7)(6)	-1
$\sigma^5\tau$	3 2 1 7 6 5 4	(1 3)(2)(4 7)(5 6)	-1
$\sigma^6\tau$	2 1 7 6 5 4 3	(1 2)(3 7)(4 6)(5)	-1

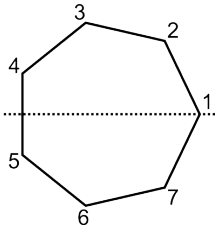

Figure 2.4.: Regular Heptagon

2.3.3.4. Linear Representations and Characters

We distinguish two cases for the dihedral group D_{2n} of order $2n$: either n is odd or even. The irreducible representations of D_{2n} are well known (e.g. see [Ser77]):

- n **is even.** Representations of degree 1:

$$\rho_0 : D_{2n} \to \mathrm{GL}(1,\mathbb{C}) : \sigma^k\tau^c \mapsto 1,$$

$$\rho_{-1} : D_{2n} \to \mathrm{GL}(1,\mathbb{C}) : \sigma^k\tau^c \mapsto \begin{cases} 1, & \text{if } c = 0, \\ -1, & \text{if } c = 1, \end{cases}$$

$$\rho_{-2} : D_{2n} \to \mathrm{GL}(1,\mathbb{C}) : \sigma^k\tau^c \mapsto (-1)^k,$$

$$\rho_{-3} : D_{2n} \to \mathrm{GL}(1,\mathbb{C}) : \sigma^k\tau^c \mapsto \begin{cases} (-1)^k, & \text{if } c = 0, \\ (-1)^{k+1}, & \text{if } c = 1. \end{cases}$$

Additionally, there are representations of degree 2. Let $\zeta_n = e^{\frac{2\pi i}{n}}$ a primitive nth root of unity and $h \in \{1, \ldots, n-1\}$.

$$\rho_h : D_{2n} \to \mathrm{GL}(2,\mathbb{C}) : \sigma^k\tau^c \mapsto \begin{cases} \begin{pmatrix} \zeta_n^{hk} & 0 \\ 0 & \zeta_n^{-hk} \end{pmatrix}, & \text{if } c = 0, \\ \begin{pmatrix} 0 & \zeta_n^{-hk} \\ \zeta_n^{hk} & 0 \end{pmatrix}, & \text{if } c = 1. \end{cases}$$

Note that ρ_h and ρ_{n-h} yield the same character, so we can actually restrict $0 < h < \frac{n}{2}$ (for $h = 0$ and $h = \frac{n}{2}$, the representations are reducible). With these, we have

$$\sum_{\chi_i} \chi_i(1)^2 = \chi_{\rho_0}(1)^2 + \chi_{\rho_{-1}}(1)^2 + \chi_{\rho_{-2}}(1)^2 + \chi_{\rho_{-3}}(1)^2 + \sum_{h=1}^{\frac{n}{2}-1} \chi_{\rho_h}(1)^2$$

$$= 1^2 + 1^2 + 1^2 + 1^2 + \sum_{h=1}^{\frac{n}{2}-1} 2^2$$
$$= 4 + (\frac{n}{2} - 1) \cdot 4 = 2n = |D_{2n}|,$$

i.e. the list of irreducible representations is complete.

- n **is odd.** Representations of degree 1:

$$\rho_0 : D_{2n} \to \mathrm{GL}(1, \mathbb{C}) : \sigma^k \tau^c \mapsto 1,$$
$$\rho_{-1} : D_{2n} \to \mathrm{GL}(1, \mathbb{C}) : \sigma^k \tau^c \mapsto \begin{cases} 1, & \text{if } c = 0, \\ -1, & \text{if } c = 1. \end{cases}$$

The representations ρ_h of degree 2 are the same as in the even case above, with $0 < h \leq \frac{n-1}{2}$. With these, we have

$$\sum_{\chi_i} \chi_i(1)^2 = \chi_{\rho_0}(1)^2 + \chi_{\rho_{-1}}(1)^2 + \sum_{h=1}^{\frac{n-1}{2}} \chi_{\rho_h}(1)^2$$
$$= 1^2 + 1^2 + \sum_{h=1}^{\frac{n-1}{2}} 2^2$$
$$= 2 + \frac{n-1}{2} \cdot 4 = 2n = |D_{2n}|,$$

i.e. the list of irreducible representations is complete.

2.3.4. Generalized Quaternion Group

Let $2 \leq n \in \mathbb{N}$. The *generalized quaternion group* of order $4n$ is the group presented by

$$Q_{4n} := \left\langle \sigma, \tau \mid \sigma^{2n} = \tau^4 = \mathrm{id},\, \tau^2 = \sigma^n,\, \tau^{-1}\sigma\tau = \sigma^{-1} \right\rangle,$$

as in [Joh80]. We observe $\sigma^k \tau^3 = \sigma^{k+n}\tau$. Thus every element $g \in Q_{4n}$ can be written uniquely in the form $g = \sigma^k \tau^b$ with $0 \leq k < 2n$ and $b \in \{0, 1\}$.

We call an element $\sigma^k \tau^b$ *imaginary*, if $b = 1$.

Lemma 2.3. *Let $G = Q_{4 \cdot 2^n}$ $(n \in \mathbb{N})$ and $H = D_{2 \cdot 2^n}$. Define $\varphi : G \to H : \sigma^k \tau^b \mapsto \sigma^{k \bmod 2^n} \tau^b$ (note that in the input element the σ and τ symbols are used to denote an element of the generalized quaternion group, whereas in the output element the σ and τ symbols specify the rotation and reflection components of a dihedral group element as defined in Section 2.3.3.1).*

Then φ is a group homomorphism, $\ker(\varphi) = \{\mathrm{id}, \sigma^{2^n}\}$, $\varphi(g) = \varphi(\sigma^{2^n} \cdot g)$ for all $g \in G$, and $G / \left\langle \sigma^{2^n} \right\rangle \cong H$.

Proof. Let $g_1 = \sigma^k \tau^b, g_2 = \sigma^l \tau^c \in G$.

- If $b = 0$:

$$\begin{aligned}\varphi(g_1 \cdot g_2) &= \varphi(\sigma^k \cdot \sigma^l \tau^c) = \varphi(\sigma^{k+l} \tau^c) = \sigma^{k+l \bmod 2^n} \tau^c \\ &= \sigma^{k \bmod 2^n} \cdot \sigma^{l \bmod 2^n} \tau^c = \varphi(g_1) \cdot \varphi(g_2).\end{aligned}$$

- If $b = 1$ and $c = 0$:

$$\begin{aligned}\varphi(g_1 \cdot g_2) &= \varphi(\sigma^k \tau \cdot \sigma^l) = \varphi(\sigma^{k-l} \tau) = \sigma^{k-l \bmod 2^n} \tau \\ &= \sigma^{k \bmod 2^n} \cdot \sigma^{-l \bmod 2^n} \tau = \sigma^{k \bmod 2^n} \tau \cdot \sigma^{l \bmod 2^n} = \varphi(g_1) \cdot \varphi(g_2).\end{aligned}$$

- If $b = 1$ and $c = 1$:

$$\begin{aligned}\varphi(g_1 \cdot g_2) &= \varphi(\sigma^k \tau \cdot \sigma^l \tau) = \varphi(\sigma^{k-l+2^n}) = \sigma^{k-l+2^n \bmod 2^n} \\ &= \sigma^{k \bmod 2^n} \cdot \sigma^{-l \bmod 2^n} \cdot \mathrm{id} = \sigma^{k \bmod 2^n} \cdot \sigma^{-l \bmod 2^n} \cdot \tau\tau \\ &= \sigma^{k \bmod 2^n} \tau \cdot \sigma^{l \bmod 2^n} \tau = \varphi(g_1) \cdot \varphi(g_2).\end{aligned}$$

Thus φ indeed is a group homomorphism.

Let $g = \sigma^k \tau^b \in G$.

$$\varphi(g) = \mathrm{id} \Leftrightarrow \varphi(\sigma^k \tau^b) = \mathrm{id} \Leftrightarrow \sigma^{k \bmod 2^n} \tau^b = \sigma^0 \tau^0 \Leftrightarrow (k \bmod 2^n = 0 \ \wedge \ b = 0).$$

As $k \bmod 2^n = 0$ (with $0 \leq k < 2 \cdot 2^n$) is fulfilled only by $k = 0$ and $k = 2^n$, we get $\ker(\varphi) = \{\mathrm{id}, \sigma^{2^n}\}$.

With this, we immediately get $\varphi(\sigma^{2^n} \cdot g) = \varphi(\sigma^{2^n}) \cdot \varphi(g) = \mathrm{id} \cdot \varphi(g) = \varphi(g)$. □

2.3.5. Groups G of Order $|G| = p^{n+1}$ with a Cyclic $N \trianglelefteq G$ of Order $|N| = p^n$

Let $p \in \mathbb{P}_{\geq 3}$, $n \geq 2$ and G a non-abelian group of order $|G| = p^{n+1}$ with a cyclic normal subgroup $N = \langle \sigma \rangle \trianglelefteq G$ of order $|N| = p^n$. Then according to [Hup67] (Theorem 14.9a) G is isomorphic to

$$\mathfrak{G}_{p \cdot p^n} = \left\langle \sigma, \tau \mid \sigma^{p^n} = \tau^p = \mathrm{id}, \tau^{-1} \sigma \tau = \sigma^{1+p^{n-1}} \right\rangle.$$

Let $n \geq 3$ and G a non-abelian group of order $|G| = 2^{n+1}$ with a cyclic normal subgroup $N = \langle \sigma \rangle \trianglelefteq G$ of order $|N| = 2^n$. Then according to [Hup67] (Theorem 14.9b) G is isomorphic to one of the following four groups (i.e. the following list is complete, and the four groups are pairwise non-isomorphic):

- Dihedral group

$$D_{2 \cdot 2^n} = \left\langle \sigma, \tau \mid \sigma^{2^n} = \tau^2 = \mathrm{id}, \tau^{-1} \sigma \tau = \sigma^{-1} \right\rangle.$$

- Generalized quaternion group

$$Q_{4 \cdot 2^{n-1}} = \left\langle \sigma, \tau \mid \sigma^{2^n} = \tau^4 = \mathrm{id}, \tau^2 = \sigma^{2^{n-1}}, \tau^{-1} \sigma \tau = \sigma^{-1} \right\rangle.$$

- Other maximal cyclic group

$$\mathfrak{G}_{2\cdot 2^n} = \left\langle \sigma, \tau \mid \sigma^{2^n} = \tau^2 = \mathrm{id}, \tau^{-1}\sigma\tau = \sigma^{1+2^{n-1}} \right\rangle.$$

- Quasi-dihedral group

$$\mathfrak{D}_{2\cdot 2^n} = \left\langle \sigma, \tau \mid \sigma^{2^n} = \tau^2 = \mathrm{id}, \tau^{-1}\sigma\tau = \sigma^{-1+2^{n-1}} \right\rangle.$$

The groups $D_{2\cdot 2^n}$, $\mathfrak{G}_{2\cdot 2^n}$ and $\mathfrak{D}_{2\cdot 2^n}$ are semidirect products (of the cyclic group $\langle\sigma\rangle$ of order 2^n with the cyclic group $\mathbb{Z}_2$, based on the automorphisms $d : \sigma \mapsto \sigma^{-1}$, $\mathfrak{g} : \sigma \mapsto \sigma^{1+2^{n-1}}$, $\mathfrak{d} : \sigma \mapsto \sigma^{-1+2^{n-1}}$), whereas $Q_{4\cdot 2^{n-1}}$ is not a semidirect product.

Lemma 2.4. *Let $n \geq 3$ and $G \in \{D_{2\cdot 2^n}, Q_{4\cdot 2^{n-1}}, \mathfrak{D}_{2\cdot 2^n}\}$. Define*

$$\varphi_l : G \to G : \sigma^k\tau^b \mapsto \sigma^{k+bl}\tau^b \text{ (with } k \in \mathbb{Z}_{2^n} \text{ and } b \in \{0,1\}\text{)}.$$

For $G = D_{2\cdot 2^n}$ and $G = Q_{4\cdot 2^{n-1}}$, φ_l is an automorphism for all $l \in \mathbb{Z}_{2^n}$. For $G = \mathfrak{D}_{2\cdot 2^n}$, φ_l is an automorphism for all even $l \in \mathbb{Z}_{2^n}$.

Proof. This is well known, but for the convenience of the reader we provide a proof.

For $D_{2\cdot 2^n}$, the result has been proven in Lemma 2.1 (we have $\varphi_l = \varphi_{1,l}$). For $Q_{4\cdot 2^{n-1}}$, the proof works similarly (with $m = 1$; interpret the elements and the automorphism over $Q_{4\cdot 2^{n-1}}$; in the $v = 1$ and $y = 1$ case the insertion of $\tau\tau$ replaces the $\sigma^{2^{n-1}}$ from the first merge).

Let $G = \mathfrak{D}_{2\cdot 2^n}$. It is clear that φ_l is bijective. Let $g = \sigma^u\tau^v, h = \sigma^x\tau^y \in G$ with $u, x \in \mathbb{Z}_{2^n}$ and $v, y \in \{0,1\}$.

- If $v = 0$:

$$\begin{aligned}\varphi_l(gh) &= \varphi_l(\sigma^u\sigma^x\tau^y) = \varphi_l(\sigma^{u+x}\tau^y) = \sigma^{u+x+yl}\tau^y = \sigma^u\sigma^{x+yl}\tau^y \\ &= \varphi_l(\sigma^u)\cdot\varphi_l(\sigma^x\tau^y) = \varphi_l(g)\cdot\varphi_l(h).\end{aligned}$$

- If $v = 1$ and $y = 0$:

$$\begin{aligned}\varphi_l(gh) &= \varphi_l(\sigma^u\tau\sigma^x) = \varphi_l(\sigma^{u-x+x\cdot 2^{n-1}}\tau) = \sigma^{u-x+x\cdot 2^{n-1}+l}\tau = \sigma^{u+l}\tau\sigma^x \\ &= \varphi_l(\sigma^u\tau)\cdot\varphi_l(\sigma^x) = \varphi_l(g)\cdot\varphi_l(h).\end{aligned}$$

- If $v = 1$ and $y = 1$:

$$\begin{aligned}\varphi_l(gh) &= \varphi_l(\sigma^u\tau\sigma^x\tau) = \varphi_l(\sigma^{u-x+x\cdot 2^{n-1}}) = \sigma^{u-x+x\cdot 2^{n-1}} = \sigma^u\tau\sigma^x\tau \\ &= \sigma^u\tau\sigma^{-l}\sigma^l\sigma^x\tau = \sigma^{u+l+l\cdot 2^{n-1}}\tau\sigma^{x+l}\tau \overset{*}{=} \sigma^{u+l}\tau\sigma^{x+l}\tau \\ &= \varphi_l(\sigma^u\tau)\cdot\varphi_l(\sigma^x\tau) = \varphi_l(g)\cdot\varphi_l(h).\end{aligned}$$

(*) l is even $\Rightarrow l\cdot 2^{n-1} \equiv 0 \pmod{2^n}$. □

Lemma 2.5. *Let $G = \mathfrak{G}_{2\cdot 2^n}$ with $n \geq 3$. Then*

$$\varphi_{l,c} : G \to G : \sigma^k\tau^b \mapsto (\sigma^l\tau^c)^k\tau^b \text{ (with } k \in \mathbb{Z}_{2^n} \text{ and } b \in \{0,1\})$$

is an automorphism for all odd $l \in \mathbb{Z}_{2^n}$ and $c \in \{0,1\}$.

Proof. This is well known, but for the convenience of the reader we provide a proof.

First we show that $\varphi_{l,c}$ is bijective. If $c = 0$, $\varphi_{l,c}$ is obviously bijective (because multiplying σ exponents with an odd number simply permutes the exponents). So, now let $c = 1$.

- Let k be even. We get $(\sigma^l\tau^c)^k\tau^b = \sigma^{l\cdot k}\sigma^{2^{n-1}\cdot l\cdot\frac{k}{2}\cdot c}\tau^b = \sigma^{l\cdot(k+2^{n-1}\cdot\frac{k}{2}\cdot c)}\tau^b$.
  - If $4 \mid k$: $k \mapsto lk \bmod 2^n$ is bijective on $M_1 := \{x \in \mathbb{Z}_{2^n} \mid \gcd(x, 2^n) = \gcd(k, 2^n)\}$.
  - If $4 \nmid k$: $k \mapsto l(k + 2^{n-1}) \bmod 2^n$ is bijective on $M_2 := \{x \in \mathbb{Z}_{2^n} \mid \gcd(x, 2^n) = 2\}$.

  Observe $\{x \in \mathbb{Z}_{2^n} \mid \gcd(x, 2^n) \geq 2\} = M_1 \dot\cup M_2$, i.e. $\varphi_{l,c}$ bijectively maps elements with even σ powers to elements with even σ powers.

- Let k be odd. We get $(\sigma^l\tau^c)^k\tau^b = \sigma^{l(k-1)}\sigma^{2^{n-1}\cdot l\cdot\frac{k-1}{2}\cdot c}\sigma^l\tau^c\tau^b = \sigma^{l(k-1+2^{n-1}\cdot\frac{k-1}{2}\cdot c+1)}\tau^{c+b} = \sigma^{l(k+2^{n-1}\cdot\frac{k-1}{2}\cdot c)}\tau^{c+b}$.
  - If $4 \mid k-1$, $\varphi_{l,c}$ maps the power of σ using $k \mapsto lk \bmod 2^n$. k can be written as $k = 4x + 1$ with some $x \in \mathbb{N}_0$. Thus, the set of all σ powers of the images of $\varphi_{l,c}$ is $M_1 := \{l(4x+1) \bmod 2^n \mid x \in \mathbb{N}_0\} = \{4x + 1 \bmod 2^n \mid x \in \mathbb{N}_0\}$.
  - If $4 \nmid k-1$, $\varphi_{l,c}$ maps the power of σ using $k \mapsto l(k + 2^{n-1}) \bmod 2^n$. k can be written as $k = 4x + 3$ with some $x \in \mathbb{N}_0$. Thus, the set of all σ powers of the images of $\varphi_{l,c}$ is $M_2 := \{l(4x + 3 + 2^{n-1}) \bmod 2^n \mid x \in \mathbb{N}_0\} = \{4x + 3 + 2^{n-1} \bmod 2^n \mid x \in \mathbb{N}_0\}$.

  Observe that $4x + 1 \equiv 4y + 3 + 2^{n-1} \pmod{2^n} \Leftrightarrow 4(x - y) \equiv 2 + 2^{n-1} \pmod{2^n}$ has no solutions, i.e. $M_1 \cap M_2 = \emptyset$. So, $\{x \in \mathbb{Z}_{2^n} \mid \gcd(x, 2^n) = 1\} = M_1 \dot\cup M_2$, i.e. $\varphi_{l,c}$ bijectively maps elements with odd σ powers to elements with odd σ powers.

Thus all in all $\varphi_{l,c}$ is bijective.

Let $g = \sigma^u\tau^v, h = \sigma^x\tau^y \in G$ with $u, x \in \mathbb{Z}_{2^n}$ and $v, y \in \{0,1\}$.

- If $v = 0$:

$$\begin{aligned}\varphi_{l,c}(gh) = \varphi_{l,c}(\sigma^u\sigma^x\tau^y) = \varphi_{l,c}(\sigma^{u+x}\tau^y) = (\sigma^l\tau^c)^{u+x}\tau^y = (\sigma^l\tau^c)^u(\sigma^l\tau^c)^x\tau^y \\ = \varphi_{l,c}(\sigma^u)\cdot\varphi_{l,c}(\sigma^x\tau^y) = \varphi_{l,c}(g)\cdot\varphi_{l,c}(h).\end{aligned}$$

- If $v = 1$ and $y = 0$:

$$\varphi_{l,c}(gh) = \varphi_{l,c}(\sigma^u\tau\sigma^x) = \varphi_{l,c}(\sigma^{u+x+x\cdot 2^{n-1}}\tau) = (\sigma^l\tau^c)^{u+x+x\cdot 2^{n-1}}\tau$$

$$
\begin{aligned}
&= (\sigma^l\tau^c)^u(\sigma^l\tau^c)^x(\sigma^l\tau^c)^{x\cdot 2^{n-1}}\tau = (\sigma^l\tau^c)^u\tau(\sigma^{l+l\cdot 2^{n-1}}\tau^c)^x(\sigma^{l+l\cdot 2^{n-1}}\tau^c)^{x\cdot 2^{n-1}}\\
&= (\sigma^l\tau^c)^u\tau(\sigma^{l+2^{n-1}}\tau^c)^x(\sigma^{l+2^{n-1}}\tau^c)^{x\cdot 2^{n-1}}\\
&= (\sigma^l\tau^c)^u\tau(\sigma^l\tau^c)^x\sigma^{x\cdot 2^{n-1}}\sigma^{x\cdot 2^{n-1}\cdot 2^{n-1}}(\sigma^l\tau^c)^{x\cdot 2^{n-1}}\\
&= (\sigma^l\tau^c)^u\tau(\sigma^l\tau^c)^x\sigma^{x\cdot 2^{n-1}}(\sigma^l\tau^c)^{x\cdot 2^{n-1}}\\
&= (\sigma^l\tau^c)^u\tau(\sigma^l\tau^c)^x\sigma^{x\cdot 2^{n-1}}\sigma^{l\cdot x\cdot 2^{n-1}}\sigma^{l\cdot 2^{n-1}\cdot\frac{x\cdot 2^{n-1}}{2}}\\
&= (\sigma^l\tau^c)^u\tau(\sigma^l\tau^c)^x = \varphi_{l,c}(\sigma^u\tau)\cdot\varphi_{l,c}(\sigma^x) = \varphi_{l,c}(g)\cdot\varphi_{l,c}(h).
\end{aligned}
$$

- If $v = 1$ and $y = 1$:

$$
\begin{aligned}
\varphi_{l,c}(gh) &= \varphi_{l,c}(\sigma^u\tau\sigma^x\tau) = \varphi_{l,c}(\sigma^{u+x+x\cdot 2^{n-1}}) = (\sigma^l\tau^c)^{u+x+x\cdot 2^{n-1}}\\
&= (\sigma^l\tau^c)^u\tau\tau(\sigma^l\tau^c)^x(\sigma^l\tau^c)^{x\cdot 2^{n-1}} = (\sigma^l\tau^c)^u\tau(\sigma^{l+l\cdot 2^{n-1}}\tau^c)^x(\sigma^{l+l\cdot 2^{n-1}}\tau^c)^{x\cdot 2^{n-1}}\tau\\
&= (\sigma^l\tau^c)^u\tau(\sigma^{l+2^{n-1}}\tau^c)^x(\sigma^{l+2^{n-1}}\tau^c)^{x\cdot 2^{n-1}}\tau\\
&= (\sigma^l\tau^c)^u\tau(\sigma^l\tau^c)^x\sigma^{x\cdot 2^{n-1}}\sigma^{x\cdot 2^{n-1}\cdot 2^{n-1}}(\sigma^l\tau^c)^{x\cdot 2^{n-1}}\tau\\
&= (\sigma^l\tau^c)^u\tau(\sigma^l\tau^c)^x\sigma^{x\cdot 2^{n-1}}(\sigma^l\tau^c)^{x\cdot 2^{n-1}}\tau\\
&= (\sigma^l\tau^c)^u\tau(\sigma^l\tau^c)^x\sigma^{x\cdot 2^{n-1}}\sigma^{l\cdot x\cdot 2^{n-1}}\sigma^{l\cdot 2^{n-1}\cdot\frac{x\cdot 2^{n-1}}{2}}\tau\\
&= (\sigma^l\tau^c)^u\tau(\sigma^l\tau^c)^x\tau = \varphi_{l,c}(\sigma^u\tau)\cdot\varphi_{l,c}(\sigma^x\tau) = \varphi_{l,c}(g)\cdot\varphi_{l,c}(h).
\end{aligned}
$$

□

2.3.6. Wreath Product

Let H be a group and P a permutation group acting on a set $\Omega = \{\omega_1, \omega_2, \ldots, \omega_{|\Omega|}\}$. Let $H^{|\Omega|} = H \times H \times \ldots \times H$ ($|\Omega|$ times) with component-wise multiplication. The *wreath product* of H with P, denoted by $H \wr_\Omega P$, is the set $H^{|\Omega|} \times P$ with the multiplication

$$(v, p) \cdot (v', p') := (v \cdot p(v'), p \cdot p')$$

where

$$p(v') = p((v'_{\omega_1}, v'_{\omega_2}, \ldots, v'_{\omega_{|\Omega|}})) := (v'_{p^{-1}(\omega_1)}, v'_{p^{-1}(\omega_2)}, \ldots, v'_{p^{-1}(\omega_{|\Omega|})}).$$

This way P defines an automorphism on $H^{|\Omega|}$, and $H \wr_\Omega P$ is the semidirect product $H^{|\Omega|} \rtimes P$.

We have $|H \wr_\Omega P| = |H|^{|\Omega|} \cdot |P|$.

A wreath product is called *regular*, if $\Omega = P$ (the group P acts on itself by left multiplication). In such a case, we omit the set that P is acting on (namely itself), i.e. $H \wr P := H \wr_P P$.

When $P \leq \mathrm{Sym}(X)$ acts on a set $X := \{1, 2, \ldots, n\}$, we write $H \wr_n P := H \wr_{\{1,2,\ldots,n\}} P$.

Subgroups. Let $G := H \wr_\Omega P$.

- $H^{|\Omega|} \times \{\mathrm{id}\}$ is a normal subgroup of G and is called the *base* of G.

- More generally, let $N \trianglelefteq H$, then $N^{|\Omega|} \times \{\mathrm{id}\}$ is a normal subgroup of G, and

  $$\varphi : G \to (H/N) \wr_\Omega P : ((v_{\omega_1}, v_{\omega_2}, \ldots, v_{\omega_{|\Omega|}}), p) \mapsto ((v_{\omega_1}N, v_{\omega_2}N, \ldots, v_{\omega_{|\Omega|}}N), p)$$

  is a surjective group homomorphism with $|\ker(\varphi)| = |N|^{|\Omega|}$.

- For $L \leq H$ define

  $$D(L) := \{(\underbrace{(h, h, \ldots, h)}_{|\Omega| \text{ times}}, \mathrm{id}) \mid h \in L\}.$$

  We have $D(L) \leq G$ and $|D(L)| = |L|$.

  If $L \leq Z(H)$, then $D(L) \trianglelefteq G$.

  For many wreath products, we have $Z(G) = D(Z(H))$. If $H = \{\mathrm{id}\}$, then $Z(G) = \{\mathrm{id}\}^{|\Omega|} \times Z(P)$.

- $\{\mathrm{id}\} \times P$ is a subgroup of G.

2.4. Set Properties

A subset $A \subseteq G$ of a group G is called *canonical*, if $\mathrm{id} \in A$.

A subset $A \subseteq G$ is said to be a *cyclic subset*, if there exists an $a \in G$ such that $A = \{\mathrm{id}, a, a^2, \ldots, a^{t-1}\}$ for some $t \in \mathbb{N}$. We do not enforce $t \geq |\langle a \rangle|$; so a cyclic subset is not necessarily a cyclic subgroup.

Let $A, B \subseteq G$ canonical with $|A \cdot B| = |A| \cdot |B|$. We call (A, B) *size-permutable*, if there exist canonical $A', B' \subseteq G$ with $|A| = |A'|$, $|B| = |B'|$ and $A \cdot B = B' \cdot A'$.

For a multiset $A = [a_1, a_2, \ldots, a_k]$ of elements $a_i \in G$, let $\mathfrak{c}_g(A)$ be the multiplicity of g in A.

For example, for $A = [1, 1, 2, 3, 4, 4, 4]$, we have $\mathfrak{c}_1(A) = 2$ and $\mathfrak{c}_4(A) = 3$.

2.4.1. Periodicity

Let $A \subseteq G$. A is called *r-periodic*, if there exists a $g \in G \setminus \{\mathrm{id}\}$ with $Ag = A$. Such a g is called an *r-period* of A. If $gA = A$ for $g \neq \mathrm{id}$, g is an *ℓ-period* and A is *ℓ-periodic*.

If A is r- or ℓ-periodic, it is also called *periodic*. If A is not periodic, it is called *aperiodic*.

In case G is abelian, r- and ℓ-periods are equivalent, thus we just speak of *periods*.

Example 2.6. If G is non-abelian, r- and ℓ-periods are not equivalent. For example, let $G = D_{2 \cdot 8}$ and $A = \{\mathrm{id}, \sigma, \tau, \sigma\tau\}$. Then the set of r-periods of A is $\{\tau\}$, and the set of ℓ-periods of A is $\{\sigma\tau\}$.

Also, a set $A \subseteq G$ can be r-periodic, but not ℓ-periodic (and the other way around). For example, let $G = D_{2 \cdot 8}$ and $A = \{\mathrm{id}, \sigma, \tau, \sigma^2\tau, \sigma^3\tau, \sigma^6\}$. Then A is r-periodic with periods $\{\sigma^2\tau\}$, but not ℓ-periodic (there is no $g \in G \setminus \{\mathrm{id}\}$ with $gA = A$).

Lemma 2.7. *Let $A \subseteq G$, $S^\diamond := \{g_1, g_2, \ldots, g_s\}$ the set of all r-periods of A, and $S := \{\mathrm{id}\} \cup S^\diamond$. Then $S \leq G$, and A is a union of cosets of S, i.e. there exists a $B \subseteq G$ such that $A = BS$ and $|A| = |B| \cdot |S|$.*

Proof. We have $S \neq \emptyset$, because $\mathrm{id} \in S$. Let $g, h \in S$. Then $gh \in S$, because $Agh = (Ag)h = Ah = A$ (i.e. gh is an r-period of A and thus in S). Also, $g^{-1} \in S$, because by multiplying $A = Ag$ by g^{-1} from the right on both sides we get $Ag^{-1} = A$. So, $S \leq G$.

G is the disjoint union of all cosets of S (the cosets form a partition of G). Pick an arbitrary $a \in A$. For every $s \in S$ we have $as \in A$ (because s is an r-period of A), i.e. $aS \subseteq A$. So, given a coset tS of S (with $t \in G$), tS is either contained completely in A or it is disjoint to A; A is a (disjoint) union of cosets of S. In order to construct B, pick one representative from each coset of S contained in A. □

2.4.2. Antiperiodicity

We now define and analyze a mapping $\mathfrak{Z}$ that extracts the "antiperiods" of a group subset.

In Section 9.1.2 we show a connection between antiperiodicity and size-permutability for dihedral groups (and based on this, we design Algorithm 9.9 for generating logarithmic signatures of dihedral groups).

Definition 2.8. Let G be a group. Define

$$\mathfrak{Z} : \mathcal{P}(G) \to \mathcal{P}(G) : R \mapsto \{z \in R \mid z \cdot R^{-1} = R\}$$

(where $z \cdot R^{-1} = \{z \cdot r^{-1} \mid r \in R\}$). By multiplying z^{-1} from the left to both sides of the equation, we obtain the following equivalent definition:

$$\mathfrak{Z} : \mathcal{P}(G) \to \mathcal{P}(G) : R \mapsto \{z \in R \mid R^{-1} = z^{-1} \cdot R\}.$$

Clearly, $\mathfrak{Z}(R) \subseteq R$.

Proposition 2.9. *Let G be a group and $U \leq G$, then $\mathfrak{Z}(U) = U$.*

Proof. For all $z, u \in U$, we get $z \cdot u^{-1} \in U$ (i.e. $z \in \mathfrak{Z}(U)$ by definition of $\mathfrak{Z}$). □

Proposition 2.10. *Let G be a group and $R \subseteq G$ with $\mathfrak{Z}(R) \neq \emptyset$, then* $\mathrm{id} \in R$.

Proof. Let $z \in \mathfrak{Z}(R)$ (at least one such z exists since $\mathfrak{Z}(R) \neq \emptyset$). By definition of $\mathfrak{Z}$, $z \cdot R^{-1} = R$ holds. As $\mathfrak{Z}(R) \subseteq R$, we have $z^{-1} \in R^{-1}$, and thus $\mathrm{id} = z \cdot z^{-1} \in R$. □

Proposition 2.11. *Let G be a group and $R \subseteq G$. If* $\mathrm{id} \in \mathfrak{Z}(R)$, *then $\mathfrak{Z}(R) \setminus \{\mathrm{id}\}$ is the set of all ℓ-periods of R.*

Proof. As $\mathfrak{Z}(R) \neq \emptyset$, by Proposition 2.10 we have $\mathrm{id} \in R$, thus all ℓ-periods of R must be elements of R. As $\mathrm{id} \in \mathfrak{Z}(R)$, $\mathrm{id} \cdot R^{-1} = R$ holds, i.e. $R^{-1} = R$. Thus $\mathfrak{Z}(R) = \{z \in R \mid z \cdot R^{-1} = R\} = \{z \in R \mid z \cdot R = R\}$. After removing id from $\mathfrak{Z}(R)$, precisely the ℓ-periods of R remain. □

Definition 2.12. Let G be a group and $R \subseteq G$. Define

$$\widehat{\mathfrak{Z}} : \mathcal{P}(G) \to \mathcal{P}(G) : R \mapsto \mathfrak{Z}(R) \cdot \mathfrak{Z}(R)^{-1}$$

(where $\mathfrak{Z}(R) \cdot \mathfrak{Z}(R)^{-1} = \{r \cdot s^{-1} \mid r, s \in \mathfrak{Z}(R)\}$).

Observe that $\widehat{\mathfrak{Z}}(R) \subseteq R$ holds (by definition of $\mathfrak{Z}$).

For arbitrary subsets $\emptyset \neq A \subseteq G$, the product $A \cdot A^{-1}$ is not necessarily a subgroup of G. For example, let $G = \mathbb{Z}_{13}$ and $A = \{0, 1\}$, then $A - A = \{0, 1, 12\}$ is not a subgroup (because for example $1 + 1 = 2 \notin A - A$). However, if $A \cdot A^{-1} = A$, then clearly $A \leq G$.

Theorem 2.13. *Let G be a group and $R \subseteq G$ with $\mathfrak{Z}(R) \neq \emptyset$.*
Then $\widehat{\mathfrak{Z}}(R) \leq G$, and $\mathfrak{Z}(R)$ is a right coset of $\widehat{\mathfrak{Z}}(R)$.

Proof. As $\mathfrak{Z}(R) \neq \emptyset$, we have $\widehat{\mathfrak{Z}}(R) \neq \emptyset$.

Let $u, v \in \widehat{\mathfrak{Z}}(R)$; we show that $u \cdot v^{-1} \in \widehat{\mathfrak{Z}}(R)$. Write $u = a \cdot b^{-1}$ and $v = c \cdot d^{-1}$ with $a, b, c, d \in \mathfrak{Z}(R)$. We have

$$u \cdot v^{-1} = a \cdot b^{-1} \cdot (c \cdot d^{-1})^{-1} = a \cdot b^{-1} \cdot d \cdot c^{-1} = a \cdot (c \cdot d^{-1} \cdot b)^{-1}.$$

For all $r \in R$, we have $b \cdot r^{-1} \in R$ (due to $b \in \mathfrak{Z}(R)$), thus $d^{-1} \cdot (b \cdot r^{-1}) \in R^{-1}$ (due to $d \in \mathfrak{Z}(R)$), thus $c \cdot (d^{-1} \cdot b \cdot r^{-1}) \in R$ (due to $c \in \mathfrak{Z}(R)$). As $\mathrm{id} \in R$ (by Proposition 2.10), we see that $c \cdot d^{-1} \cdot b = c \cdot d^{-1} \cdot b \cdot \mathrm{id}^{-1} \in R$. Therefore, $c \cdot d^{-1} \cdot b \in \mathfrak{Z}(R)$.

As $a \in \mathfrak{Z}(R)$ and $c \cdot d^{-1} \cdot b \in \mathfrak{Z}(R)$, we get $a \cdot (c \cdot d^{-1} \cdot b)^{-1} \in \widehat{\mathfrak{Z}}(R)$, i.e. $u \cdot v^{-1} \in \widehat{\mathfrak{Z}}(R)$.

Consequently, $\widehat{\mathfrak{Z}}(R) \leq G$.

Let $x, y, z \in \mathfrak{Z}(R)$. Observe that $x \cdot y^{-1} \cdot z \in \mathfrak{Z}(R)$, because for all $r \in R$ we have $z \cdot r^{-1} \in R$, $y^{-1} \cdot (z \cdot r^{-1}) \in R^{-1}$, $x \cdot (y^{-1} \cdot z \cdot r^{-1}) \in R$, and $x \cdot y^{-1} \cdot z \in R$ due to $\mathrm{id} \in R$. So, we have $\mathfrak{Z}(R) \cdot \mathfrak{Z}(R)^{-1} \cdot \mathfrak{Z}(R) \subseteq \mathfrak{Z}(R)$. By definition, $\widehat{\mathfrak{Z}}(R) = \mathfrak{Z}(R) \cdot \mathfrak{Z}(R)^{-1}$, so $\widehat{\mathfrak{Z}}(R) \cdot \mathfrak{Z}(R) \subseteq \mathfrak{Z}(R)$. As already proven, $\widehat{\mathfrak{Z}}(R) \leq G$, thus especially $\mathrm{id} \in \widehat{\mathfrak{Z}}(R)$, and consequently $\widehat{\mathfrak{Z}}(R) \cdot \mathfrak{Z}(R) \supseteq \mathfrak{Z}(R)$. We obtain

$$\widehat{\mathfrak{Z}}(R) \cdot \mathfrak{Z}(R) = \mathfrak{Z}(R).$$

Reading the equation from right to left, $\mathfrak{Z}(R)$ is the union of one or more right cosets of $\widehat{\mathfrak{Z}}(R)$. Two cosets of a subgroup are either the same or disjoint. As the sets on the left side and the right side of the equation must contain the same number of elements, $\mathfrak{Z}(R)$ must be exactly one right coset of $\widehat{\mathfrak{Z}}(R)$ (all elements in $\mathfrak{Z}(R)$ result in the same coset being generated together with $\widehat{\mathfrak{Z}}(R)$). □

Corollary 2.14. *Let G be a group and $R \subseteq G$. If $\mathrm{id} \in \mathfrak{Z}(R)$, then $\mathfrak{Z}(R) \leq G$.*

Proof. This directly follows from Theorem 2.13. □

Proposition 2.15. *Let G be a group and $R \subseteq G$ with $\mathfrak{Z}(R) \neq \emptyset$. Then*

$$|\mathfrak{Z}(R)| \mid |R| \quad \textit{and} \quad |\mathfrak{Z}(R)| \mid |G|\,.$$

Proof. We have $\mathfrak{Z}(R) \cdot R^{-1} = R$ (by definition of $\mathfrak{Z}$). By Theorem 2.13, $\mathfrak{Z}(R)$ is a right coset of the subgroup $\widehat{\mathfrak{Z}}(R) \leq G$. Multiplying $\mathfrak{Z}(R)$ together with an element $r \in R^{-1}$ results in another coset of the subgroup (we have $\mathfrak{Z}(R) = \widehat{\mathfrak{Z}}(R) \cdot g$ for some $g \in \mathfrak{Z}(R)$, and thus $\mathfrak{Z}(R) \cdot r = (\widehat{\mathfrak{Z}}(R) \cdot g) \cdot r = \widehat{\mathfrak{Z}}(R) \cdot (r \cdot g)$ is another right coset of $\widehat{\mathfrak{Z}}(R)$). So, $\mathfrak{Z}(R) \cdot R^{-1}$ is a union of (right) cosets of a subgroup of G. Two cosets of a subgroup are either the same or disjoint. Thus, $|R|$ is a multiple of $|\mathfrak{Z}(R)|$.

As $|\mathfrak{Z}(R)| = \left|\widehat{\mathfrak{Z}}(R)\right|$ and $\widehat{\mathfrak{Z}}(R) \leq G$, $|\mathfrak{Z}(R)| \mid |G|$ follows by Lagrange's theorem. □

Corollary 2.16. *Let $G = \mathbb{Z}_n$, $p \in \mathbb{P}$ with $p \,|\, n$, and $R \subseteq G$ with $|R| = p$. Then*

$$\exists\, z \in R : \mathfrak{Z}(R) \in \{\emptyset, \{z\}, \left\langle \frac{n}{p} \right\rangle\}.$$

Proof. The case $\mathfrak{Z}(R) = \emptyset$ is clear, thus now assume $\mathfrak{Z}(R) \neq \emptyset$. By Theorem 2.13, $\mathfrak{Z}(R)$ is a coset of a subgroup of G. By Proposition 2.15, the only possible orders for the coset are 1 and p.

The possible cosets of the subgroup $\{0\}$ are $\{z\}$ for $z \in \mathfrak{Z}(R) \subseteq R$.

If $|\mathfrak{Z}(R)| = p$, then $\mathfrak{Z}(R)$ is a coset of the subgroup $\left\langle \frac{n}{p} \right\rangle \leq G$ (generated additively), which is the only subgroup of order p (due to $\mathbb{Z}_n$ being cyclic). As $\mathfrak{Z}(R) \subseteq R$ and $|\mathfrak{Z}(R)| = p = |R|$, it must be $\mathfrak{Z}(R) = R$. As $0 \in R$ (by Proposition 2.10), we see that R is the coset containing 0, i.e. the subgroup itself. □

Proposition 2.17. *Let G be a group, $N \trianglelefteq G$, $\varphi : G \to G/N : g \mapsto gN$ the canonical projection onto G/N, and $R \subseteq G$. Then*

$$\mathfrak{Z}(\varphi(R)) \supseteq \varphi(\mathfrak{Z}(R)).$$

Proof. Let $\varphi(z) \in \varphi(\mathfrak{Z}(R))$ arbitrary (with an appropriate $z \in \mathfrak{Z}(R)$). Then $\varphi(z) \cdot \varphi(r)^{-1} \in \varphi(R)$ holds for all $r \in R$ (and thus for all $\varphi(r) \in \varphi(R)$), because

$$\begin{aligned} & z \cdot r^{-1} \in R \\ \Rightarrow\ & \varphi(z \cdot r^{-1}) \in \varphi(R) \\ \Rightarrow\ & \varphi(z) \cdot \varphi(r^{-1}) \in \varphi(R) \\ \Rightarrow\ & \varphi(z) \cdot \varphi(r)^{-1} \in \varphi(R). \end{aligned}$$

So, $\varphi(z) \in \mathfrak{Z}(\varphi(R))$, i.e. $\mathfrak{Z}(\varphi(R)) \supseteq \varphi(\mathfrak{Z}(R))$. □

Note that Proposition 2.17 is sharp; only "$\supseteq$" holds, not "$=$". For example, let $G = \mathbb{Z}_8$, $N = \{0, 4\}$, $\varphi : G \to G/N : g \mapsto g + N$, $R = \{0, 1, 3, 6\}$, then $\varphi(R) = \{0, 1, 2, 3\} = G/N$, $\mathfrak{Z}(R) = \{1\}$, $\mathfrak{Z}(\varphi(R)) = \{0, 1, 2, 3\} = G/N$, i.e. $\mathfrak{Z}(\varphi(R)) \supset \varphi(\mathfrak{Z}(R))$.

Proposition 2.18. *Let G be an abelian group and $A, B \subseteq G$ canonical with $|A + B| = |A| \cdot |B|$.*

Then for each of the following statements there exists an example where the statement holds and a counter-example where the statement does not hold (i.e. none of the statements holds in general):

- $\mathfrak{Z}(A+B)\ \mathfrak{R}\ \mathfrak{Z}(A)$ *for each* $\mathfrak{R} \in \{=, \subset, \subseteq, \supset, \supseteq\}$.
- $|\mathfrak{Z}(A+B)|\ \mathfrak{R}\ |\mathfrak{Z}(A)| + |\mathfrak{Z}(B)|$ *for each* $\mathfrak{R} \in \{=, <, \leq, >, \geq\}$.
- $|\mathfrak{Z}(A+B)|\ \mathfrak{R}\ \min\{|\mathfrak{Z}(A)|, |\mathfrak{Z}(B)|\}$ *for each* $\mathfrak{R} \in \{=, >\}$.

Proof. We give a few examples. Note that A and B can be interchanged, because G is abelian.

- Let $G = \mathbb{Z}_{27}$, $A = \{0, 1, 5\}$, $B = \{0, 9, 18\}$.
  Then $A + B = \{0, 1, 5, 9, 10, 14, 18, 19, 23\}$,
  $\mathfrak{Z}(A) = \emptyset$, $\mathfrak{Z}(B) = B$, $\mathfrak{Z}(A+B) = \{1, 10, 19\}$,
  $\mathfrak{Z}(A+B) \supset \mathfrak{Z}(A)$, $\mathfrak{Z}(A+B)\ \mathfrak{R}\ \mathfrak{Z}(B)$ is false for all $\mathfrak{R} \in \{=, \subset, \subseteq, \supset, \supseteq\}$,
  $|\mathfrak{Z}(A+B)| = |\mathfrak{Z}(A)| + |\mathfrak{Z}(B)|$, $|\mathfrak{Z}(A+B)| = \min\{|\mathfrak{Z}(A)|, |\mathfrak{Z}(B)|\}$.
- Let $G = \mathbb{Z}_{27}$, $A = \{0, 1, 11\}$, $B = \{0, 3, 15\}$ (note that $A + B + \{0, 9, 18\} = G$).
  Then $A + B = \{0, 1, 3, 4, 11, 14, 15, 16, 26\}$,
  $\mathfrak{Z}(A) = \emptyset$, $\mathfrak{Z}(B) = \{3\}$, $\mathfrak{Z}(A+B) = \emptyset$,
  $\mathfrak{Z}(A+B) = \mathfrak{Z}(A)$, $\mathfrak{Z}(A+B) \subset \mathfrak{Z}(B)$,
  $|\mathfrak{Z}(A+B)| < |\mathfrak{Z}(A)| + |\mathfrak{Z}(B)|$, $|\mathfrak{Z}(A+B)| = \min\{|\mathfrak{Z}(A)|, |\mathfrak{Z}(B)|\}$.
- Let $G = \mathbb{Z}_{27}$, $A = \{0, 3, 15\}$, $B = \{0, 9, 18\}$ (note that $A + B \leq G$).
  Then $A + B = \{0, 3, 6, 9, 12, 15, 18, 21, 24\}$,
  $\mathfrak{Z}(A) = \{3\}$, $\mathfrak{Z}(B) = B$, $\mathfrak{Z}(A+B) = A + B$,
  $\mathfrak{Z}(A+B) \supset \mathfrak{Z}(A)$, $\mathfrak{Z}(A+B) \supset \mathfrak{Z}(B)$,
  $|\mathfrak{Z}(A+B)| > |\mathfrak{Z}(A)| + |\mathfrak{Z}(B)|$, $|\mathfrak{Z}(A+B)| > \min\{|\mathfrak{Z}(A)|, |\mathfrak{Z}(B)|\}$.
- Let $G = \mathbb{Z}_{27}$, $A = \{0, 3, 15\}$, $B = \{0, 5, 16\}$.
  Then $A + B = \{0, 3, 4, 5, 8, 15, 16, 19, 20\}$,
  $\mathfrak{Z}(A) = \{3\}$, $\mathfrak{Z}(B) = \{5\}$, $\mathfrak{Z}(A+B) = \{8\}$,
  $\mathfrak{Z}(A+B)\ \mathfrak{R}\ \mathfrak{Z}(A)$ and $\mathfrak{Z}(A+B)\ \mathfrak{R}\ \mathfrak{Z}(B)$ are false for all $\mathfrak{R} \in \{=, \subset, \subseteq, \supset, \supseteq\}$,
  $|\mathfrak{Z}(A+B)| < |\mathfrak{Z}(A)| + |\mathfrak{Z}(B)|$, $|\mathfrak{Z}(A+B)| = \min\{|\mathfrak{Z}(A)|, |\mathfrak{Z}(B)|\}$. □

Proposition 2.19. *Let G be a group, $A, B \subseteq G$ with $A \subseteq C_G(B)$, and $|A \cdot B| = |A| \cdot |B|$. Then*

$$\mathfrak{Z}(A \cdot B) \supseteq \mathfrak{Z}(A) \cdot \mathfrak{Z}(B)$$

(where $\mathfrak{Z}(A) \cdot \mathfrak{Z}(B) = \{z \cdot z' \mid z \in \mathfrak{Z}(A), z' \in \mathfrak{Z}(B)\}$).

Proof. If $\mathfrak{Z}(A) = \emptyset$ or $\mathfrak{Z}(B) = \emptyset$, then the statement obviously holds (because the right side is $\emptyset$). Thus now assume that $|\mathfrak{Z}(A)| \geq 1$ and $|\mathfrak{Z}(B)| \geq 1$.

Let $z \in \mathfrak{Z}(A)$ and $z' \in \mathfrak{Z}(B)$. Then $z \cdot z' \in A \cdot B$ (because $z \in \mathfrak{Z}(A) \subseteq A$ and $z' \in \mathfrak{Z}(B) \subseteq B$). For all $g = a \cdot b \in A \cdot B$, we get

$$(z \cdot z') \cdot g^{-1} = z \cdot z' \cdot (a \cdot b)^{-1} = z \cdot z' \cdot b^{-1} \cdot a^{-1} \overset{*}{=} \underbrace{z \cdot a^{-1}}_{\in A} \cdot \underbrace{z' \cdot b^{-1}}_{\in B} \in A \cdot B.$$

(*) $(z' \cdot b^{-1}) \cdot a^{-1} = a^{-1} \cdot (z' \cdot b^{-1})$ holds due to $A^{-1} \subseteq C_G(B)$ (which holds due to $A \subseteq C_G(B)$ and $C_G(B) \leq G$).

So, $z \cdot z' \in \mathfrak{Z}(A \cdot B)$, i.e. $\mathfrak{Z}(A \cdot B) \supseteq \mathfrak{Z}(A) \cdot \mathfrak{Z}(B)$. □

Note that Proposition 2.19 is sharp; only "$\supseteq$" holds, not "=", as it can be seen from the examples in the proof of Proposition 2.18.

2.5. Efficiency

A deterministic algorithm is called a *polynomial-time algorithm*, if its run-time is bounded by a polynomial in the input length.

A randomized algorithm that always computes correct results is called a *Las Vegas algorithm*.

A Las Vegas algorithm is called a *probabilistic polynomial-time algorithm*, if the expected value of its run-time is bounded by a polynomial in the input length.

If a problem can be solved by a probabilistic polynomial-time algorithm, the problem is called *easy* and the algorithm is called *efficient*. Otherwise, the problem is called *hard*.

2.6. One-Way Functions

A *one-way function* f is a function that can easily be computed for every input, but computing $f^{-1}(a)$ for a given image a is hard in general.

More formally, a function $f : \mathbb{Z}_2^* \to \mathbb{Z}_2^*$ is called a one-way function, if:

- There exists a deterministic algorithm that for every input $x \in \mathbb{Z}_2^*$ outputs the value $f(x)$ within time polynomial in the length of x (i.e. the number of bits).
- For every probabilistic polynomial-time algorithm F and every positive polynomial $P \in \mathbb{Z}[x]$ the following holds for all sufficiently large $n \in \mathbb{N}$:

$$\Pr[F(f(x), 1^n) \in f^{-1}(f(x)) \mid x \leftarrow \Omega_n] < \frac{1}{P(n)},$$

  where "$x \leftarrow \Omega_n$" means to randomly choose an element x from the uniform distribution on $\mathbb{Z}_2^n$.

The helper input 1^n (string of n ones) ensures that F has a chance to run in time polynomial in its input length (if the 1^n would be omitted and the images of f are of length $\lceil \log_2 n \rceil$, no polynomial-time algorithm can output a preimage).

The definition ensures that f is hard to invert in the average case (not in the worst case only).

Up to now it is unknown whether one-way functions exist. P $\neq$ NP is a necessary condition for the existence of one-way functions.

A *one-way permutation* is a bijective, length-preserving one-way function.

Let $k_U : \mathbb{N} \to \mathbb{N}$ and $k_V : \mathbb{N} \to \mathbb{N}$. Let $(U_n)_{n\in\mathbb{N}}$ and $(V_n)_{n\in\mathbb{N}}$ be families with $U_n = \{0, 1, \ldots, k_U(n) - 1\}$ and $V_n = \{0, 1, \ldots, k_V(n) - 1\}$ for all $n \in \mathbb{N}$.

A function family $(f_n : U_n \to V_n)_{n\in\mathbb{N}}$ with all f_n pairwise different (and typically $|U_{n+1}| \geq |U_n|$ and $|V_{n+1}| \geq |V_n|$ for all $n \in \mathbb{N}$) is called one-way, if:

- There exists a deterministic algorithm that for every input $x \in U_n$ outputs the value $f(x)$ within time polynomial in $\log_2 |U_n|$.
- For every probabilistic polynomial-time algorithm F and every positive polynomial $P \in \mathbb{Z}[x]$ the following holds for all sufficiently large $n \in \mathbb{N}$:

$$\Pr[F(f(x), 1^{\lceil \log_2 |U_n| \rceil}) \in f^{-1}(f(x)) \mid x \leftarrow \Omega_{U_n}] < \frac{1}{P(\log_2 |U_n|)},$$

  where "$x \leftarrow \Omega_{U_n}$" means to randomly choose an element x from the uniform distribution on U_n.

Let $(f_n : U_n \to V_n)_{n\in\mathbb{N}}$ be a family of one-way functions. If all f_n are bijective, $(f_n : U_n \to V_n)_{n\in\mathbb{N}}$ is a family of one-way permutations.

Usually we omit the family notation and just call a single mapping f a one-way function/permutation, keeping in mind that actually a whole family is meant.

For example, by "let $p \in \mathbb{P}$ and $f : \mathbb{Z}_p \to \mathbb{Z}_p$ a one-way permutation" we actually mean a family $(f_n : \mathbb{Z}_{i(n)} \to \mathbb{Z}_{i(n)})_{n\in\mathbb{N}}$ of one-way permutations f_n with $i(n) \in \mathbb{P}$ for all $n \in \mathbb{N}$.

2.7. Cryptographically Secure Pseudo-Random Number Generators (CSPRNGs)

A *cryptographically secure pseudo-random number generator (CSPRNG)* is a function $G : \mathbb{Z}_2^* \to \mathbb{Z}_2^*$ with the following properties:

- There exists a function $l : \mathbb{N}_0 \to \mathbb{N}_0$ with $l(n) > n$ for all $n \in \mathbb{N}_0$ such that $G(x) \in \mathbb{Z}_2^{l(n)}$ for all $x \in \mathbb{Z}_2^n$ and $n \in \mathbb{N}_0$.
- There exists a deterministic algorithm that for every input $x \in \mathbb{Z}_2^*$ outputs the value $G(x)$ within time polynomial in the length of x (i.e. the number of bits).
- For every probabilistic polynomial-time algorithm $D : \mathbb{Z}_2^* \to \mathbb{Z}_2$ and every positive polynomial $P \in \mathbb{Z}[x]$ the following holds for all sufficiently large $n \in \mathbb{N}$:

$$\left|\Pr[D(G(x)) = 1 \mid x \leftarrow \Omega_n] - \Pr[D(z) = 1 \mid z \leftarrow \Omega_{l(n)}]\right| < \frac{1}{P(n)},$$

  where "$y \leftarrow \Omega_k$" means to randomly choose an element y from the uniform distribution on $\mathbb{Z}_2^k$.

Like with one-way functions, up to now it is unknown whether CSPRNGs exist. CSPRNGs exist if and only if one-way functions exist [Hås99].

2.8. Pseudo-Code

Most algorithms in this work are described using pseudo-code. Some notes on the pseudo-code:

- Assignments are denoted using the "$\leftarrow$" symbol. The "=" symbol denotes an equality test (not an assignment).

  For example, by "$x \leftarrow 4$" we mean to set the variable x to the value 4. The expression "$y = 3$" is true if and only if the variable y currently has the value 3.

- Indices (for lists, vectors, blocks, block sequences, etc.) are one-based.

  For example, for $v = (3, 7, 2, 9)$ we have $v[1] = 3$ and $v[4] = 9$.

- Loop boundaries are inclusive.

  For example, by "for $i \leftarrow 1$ to 4" we mean to execute the following code with $i = 1$, $i = 2$, $i = 3$ and $i = 4$ (in this order).

- The keywords "`return`", "`continue`" and "`break`" have the same meaning as in C-like languages (C++, Java, C#, etc.):
  - "`Return`" means to leave the current function (or the whole algorithm, if no function has been declared). In case the function/algorithm returns a value, this value follows the "`return`" keyword.
  - "`Continue`" means to skip the rest of the current innermost loop iteration and continue with the next iteration of the loop (including the advance of any running variables).
  - "`Break`" means to leave the innermost loop that we are currently in.

  The keywords are case-insensitive (e.g. "`Return`" and "`return`" mean exactly the same; we use the one or the other depending on where in the sentence the keyword appears).

  Example:
  - For $i \leftarrow 1$ to 8:
    1. If $i = 3$: `continue`.
    2. If $i = 6$: `break`.
    3. Output i.

  This algorithm outputs "1 2 4 5".

- The data types Int and UInt are expected to be able to store arbitrarily large numbers (so-called "big integers"). An Int stores any value from $\mathbb{Z}$, and an UInt stores any value from $\mathbb{N}_0$.

3. Group Factorizations

Chapter 3 starts by defining group factorizations and factorization mappings. We define what "tame" and "wild" mean precisely, and we specify classes of block sequences.

3.1. Definitions and Notations

Let G be a group, $s \in \mathbb{N}$, $A_1, \ldots, A_s$ finite sequences of elements of G (the A_i are called *blocks*), and $\alpha = (A_1, \ldots, A_s)$ a block sequence.

We write $|\alpha|$ for the number of blocks in α, i.e. $|\alpha| = s$.

$|A_i|$ denotes the number of elements in the block A_i (counting multiples).

The vector $t(\alpha) := (|A_1|, |A_2|, \ldots, |A_s|)$ is called the *type* of α.

The *length* $\ell(\alpha)$ of α is defined as $\ell(\alpha) := \sum_{i=1}^{s} |A_i|$.

If an element $g \in G$ is contained in a block A_i (at any position), we write "$g \in A_i$".

For the ith block A_i of α we also write $\alpha[i]$. The jth element in the ith block is denoted by $A_i[j]$ or $\alpha[i][j]$. If the context is clear, we also write $a_{i,j}$ for this element. i and j are 1-based, i.e. the first element in the first block of α is $\alpha[1][1]$.

- α is called a *cover*, if $\{a_1 \cdot a_2 \cdots a_s \mid a_1 \in A_1, a_2 \in A_2, \ldots, a_s \in A_s\} = G$, i.e. every element $g \in G$ can be written as a product $g = a_1 \cdot a_2 \cdots a_s$ with $a_1 \in A_1, a_2 \in A_2, \ldots, a_s \in A_s$.

  Generating a random cover with high probability is possible efficiently [Sva07].

- α is called a *pseudo-logarithmic signature*, if $\prod_{i=1}^{s} |A_i| = |G|$.

- α is called a *logarithmic signature*, if it is a cover and a pseudo-logarithmic signature. For every element $g \in G$ there exists exactly one way to write g as a product $g = a_1 \cdot a_2 \cdots a_s$ with $a_1 \in A_1, a_2 \in A_2, \ldots, a_s \in A_s$.

- α is called a *k-factorization*, if every element $g \in G$ can be written in exactly k different ways with respect to α (i.e. there are exactly k different ways to select one element from each block of α such that the product gives g).

- α is called an *$[s, r]$-mesh*, if α is a cover, $|A_i| = r$ for every $1 \leq i \leq s$ and

$$\frac{\max\{c_g \mid g \in G\}}{\min\{c_g \mid g \in G\}} \leq 2,$$

  where c_g denotes the number of factorizations of an element $g \in G$ with respect to α.

In this work, we mostly regard logarithmic signatures.

The term "logarithmic" comes from the fact that a logarithmic signature α contains $\sum_{i=1}^{s} |A_i|$ elements, but generates $\prod_{i=1}^{s} |A_i|$ elements.

Let $\Xi(G)$ denote the set of all finite block sequences of elements of G, $\Xi_s(G)$ the set of all block sequences in $\Xi(G)$ consisting of s blocks, $\Xi_{(r_1,\ldots,r_s)}(G) := \{(B_1, \ldots, B_s) \in \Xi_s(G) \mid |B_1| = r_1, |B_2| = r_2, \ldots, |B_s| = r_s\}$, and $\Lambda(G)$ the set of all logarithmic signatures for a group G.

Define $E(\alpha) := \bigcup_{i=1}^{s} \{A_i[j] \mid 1 \leq j \leq |A_i|\}$, the (unordered) set of all elements in the blocks of α. If α is a cover, then $\langle E(\alpha) \rangle = G$, but usually the generators $E(\alpha)$ are not independent.

Similarly, for a block A, define $E(A) := \{A[i] \mid 1 \leq i \leq |A|\}$, the (unordered) set of all elements in A. For example, if $A = (0, 2, 1, 2, 3, 0)$, then $E(A) = \{0, 1, 2, 3\}$.

We often implicitly convert between blocks and sets. During an implicit conversion from a set S to a block A, the elements in S may be arranged in an arbitrary order to form an ordered block A (such that A contains every element of S exactly once). During an implicit conversion from a block A to a set S, duplicate elements are merged and the set is unordered, i.e. $S = E(A)$.

Let A be a block. Then for instance by "id $\in A$" we mean that the identity element is contained in A; its position within A is arbitrary and id may be contained in A multiple times. Furthermore, for example by "let $A \in \mathcal{P}(G)$ a block" we assume an implicit conversion to happen from a set to a block (i.e. the elements being picked from G are put into an ordered block in an arbitrary order, but without duplicates).

In light of this, the main purpose of $E(A)$ is to remove duplicate elements from the block A (the set returned by $E(A)$ may be converted automatically to a block again).

A block A is called a subgroup of G, if $|A| = |E(A)|$ and $E(A) \leq G$.

Let $\varphi : G \to H$ be a group homomorphism, and $A = (a_1, \ldots, a_k)$ a block with $a_i \in G$ for all $1 \leq i \leq k$. We write $\varphi(A) := (\varphi(a_1), \ldots, \varphi(a_k))$ (note that A and $\varphi(A)$ are blocks and thus may contain duplicate elements; we have $|A| = |\varphi(A)|$).

Let $\alpha \in \Xi_s(G)$. We write $\varphi(\alpha) := (\varphi(\alpha[1]), \ldots, \varphi(\alpha[s])) \in \Xi_s(H)$. Observe that $t(\varphi(\alpha)) = t(\alpha)$.

A *factorization* of an element $g \in G$ is a sequence of indices $(i_1, i_2, \ldots, i_s)$ such that

$$\alpha[1][i_1] \cdot \alpha[2][i_2] \cdots \alpha[s][i_s] = g.$$

For an improved readability of block sequences, we sometimes omit the parentheses enclosing all blocks and the commas between blocks. If a block contains only one element, we may omit the block parentheses. Furthermore, we sometimes write blocks as column vectors. For example,

$$\alpha = (((0,0,0),(3,0,1)),((0,0,0),(2,0,0)),((0,0,0),(0,1,1),(0,2,0)),((0,0,0),(0,0,1)))$$

and

$$\alpha = \binom{(0,0,0)}{(3,0,1)}\binom{(0,0,0)}{(2,0,0)}\binom{(0,0,0)}{(0,1,1)}\binom{(0,0,0)}{(0,2,0)}\binom{(0,0,0)}{(0,0,1)}$$

denote exactly the same block sequence (actually a logarithmic signature) for $\mathbb{Z}_4 \oplus \mathbb{Z}_3 \oplus \mathbb{Z}_2$.

3.2. Factorization Mappings

Let $\alpha \in \Xi_s(G)$ of type $t(\alpha) = (r_1, \ldots, r_s)$. An element selection or product mapping can be defined as

$$\Theta_\alpha : \mathbb{Z}_{r_1} \oplus \mathbb{Z}_{r_2} \oplus \ldots \oplus \mathbb{Z}_{r_s} \to G : (i_1, i_2, \ldots, i_s) \mapsto \alpha[1][i_1 + 1] \cdot \alpha[2][i_2 + 1] \cdots \alpha[s][i_s + 1].$$

It is convenient to define a bijection to express the indices vector as a non-negative integer using a mixed radix representation:

$$\lambda : \mathbb{Z}_{r_1} \oplus \mathbb{Z}_{r_2} \oplus \ldots \oplus \mathbb{Z}_{r_s} \to \mathbb{Z}_{\prod_{i=1}^{s} r_i} : (i_1, i_2, \ldots, i_s) \mapsto \sum_{j=1}^{s} i_j m_j$$

with

$$m_j := \prod_{k=1}^{j-1} r_k = \begin{cases} 1, & \text{for } j = 1, \\ r_1 \cdots r_{j-1}, & \text{otherwise.} \end{cases}$$

Define another mapping by composing the above two mappings:

$$\breve{\alpha} : \mathbb{Z}_{\prod_{i=1}^{s} r_i} \to G : x \mapsto \Theta_\alpha(\lambda^{-1}(x)).$$

If α is a logarithmic signature, we have $\prod_{i=1}^{s} r_i = |G|$, and both Θ_α and $\breve{\alpha}$ are bijections.

Let $\alpha, \beta \in \Xi(G)$. If $\breve{\alpha} = \breve{\beta}$, then α and β are called *equivalent*, and we also write $\alpha \doteq \beta$.

This equivalence relation is interesting, because most cryptosystems using block sequences (we present such cryptosystems in Chapter 4) actually use the mapping $\breve{\alpha}$ instead of directly using an $\alpha \in \Xi(G)$. For these cryptosystems, two block sequences $\alpha, \beta \in \Xi(G)$ result in different encryption functions if and only if $\breve{\alpha} \neq \breve{\beta}$.

Let $\alpha, \beta \in \Lambda(G)$ with $t(\alpha) = t(\beta)$. As we will see in Proposition 5.6, $\breve{\alpha} = \breve{\beta}$ holds if and only if α is a sandwich of β (sandwiches will be defined in Section 5.1.4).

Example 3.1. Let $G = \mathbb{Z}_8$ and

$$\alpha := \binom{0}{1}\binom{0}{2}\binom{0}{4} \in \Lambda(G), \quad \beta := \binom{2}{3}\binom{1}{3}\binom{5}{1} \in \Lambda(G),$$

then $\alpha \neq \beta$, but $\breve{\alpha} = \breve{\beta}$, because

$$\beta = \binom{2}{3}\binom{1}{3}\binom{5}{1} \doteq \binom{0}{1}(2)(6)\binom{0}{2}(3)(5)\binom{0}{4} \doteq \binom{0}{1}(0)\binom{0}{2}(0)\binom{0}{4} \doteq \alpha.$$

β was constructed from α using a sandwich transformation.

Example 3.2. Let $G = \mathbb{Z}_8$ and

$$\alpha := \binom{0}{1}\binom{0}{2}\binom{0}{4},\ \beta := \binom{0}{1}\binom{0}{2}\binom{4}{0},\ \gamma := \binom{0}{1}\binom{0}{4}\binom{0}{2} \in \Lambda(G),$$

then $\breve{\alpha} \neq \breve{\beta}$, $\breve{\alpha} \neq \breve{\gamma}$ and $\breve{\beta} \neq \breve{\gamma}$, e.g. due to

$$\begin{aligned}\breve{\alpha}(3) &= \alpha[1][2] + \alpha[2][2] + \alpha[3][1] = 1 + 2 + 0 = 3,\\ \breve{\beta}(3) &= \beta[1][2] + \beta[2][2] + \beta[3][1] = 1 + 2 + 4 = 7,\\ \breve{\gamma}(3) &= \gamma[1][2] + \gamma[2][2] + \gamma[3][1] = 1 + 4 + 0 = 5,\end{aligned}$$

so $\breve{\alpha}(3) = 3 \neq 7 = \breve{\beta}(3)$, $\breve{\alpha}(3) = 3 \neq 5 = \breve{\gamma}(3)$ and $\breve{\beta}(3) = 7 \neq 5 = \breve{\gamma}(3)$.

We observe that $\breve{\alpha}$ is efficiently computable for all $\alpha \in \Xi_s(G)$ (simply computing the product of s group elements), but computing $\breve{\alpha}^{-1}$ can be hard (it requires finding factorizations of given group elements with respect to α). We state this more precisely in Section 3.3.

Two logarithmic signatures $\alpha, \beta \in \Lambda(G)$ can be composed to get a permutation on $\mathbb{Z}_{|G|}$:

$$P_{\alpha,\beta} : \mathbb{Z}_{|G|} \to \mathbb{Z}_{|G|} : x \mapsto \breve{\beta}^{-1}(\breve{\alpha}(x)).$$

Let $\eta \in \Lambda(G)$ be a fixed logarithmic signature. For $\alpha \in \Lambda(G)$ define

$$\widehat{\alpha} : \mathbb{Z}_{|G|} \to \mathbb{Z}_{|G|} : x \mapsto P_{\alpha,\eta}(x) = \breve{\eta}^{-1}(\breve{\alpha}(x)).$$

Observe that for every $x \in \mathbb{Z}_{|G|}$,

$$P_{\alpha,\beta}(x) = \breve{\beta}^{-1}(\breve{\alpha}(x)) = \breve{\beta}^{-1}(\breve{\eta}(\breve{\eta}^{-1}(\breve{\alpha}(x)))) = \widehat{\beta}^{-1}(\widehat{\alpha}(x)).$$

3.3. Tame and Wild

Encoding. Let the elements of a group G (for instance numbers, permutations, matrices or words in generators) be encoded by binary strings of uniform length b_G; b_G is called the *code length*. Not every string of $\mathbb{Z}_2^{b_G}$ needs to represent a group element.

An $\alpha \in \Xi(G)$ can be stored as a list of blocks containing group elements. We call this the *list representation* of α. Every $\alpha \in \Xi(G)$ can be described by a list representation. If not specified differently, in the following we always assume that block sequences are given by list representations.

When a pair (G, α) is an input to an algorithm, the space required to encode this pair can be estimated by

$$\mathcal{S}(G, \alpha) := b_G \cdot \ell(\alpha).$$

Here we assume that negligible space is required to encode/identify the group G.

Time. In order to measure the time complexity of an algorithm having a pair (G, α) as input, we need to regard families $(G_n, \alpha_n)_{n\in\mathbb{N}}$ with all α_n pairwise different, and determine how the run-time of the algorithm relates to the input size for $n \to \infty$.

A Las Vegas algorithm on a given family $(G_n, \alpha_n)_{n\in\mathbb{N}}$ (i.e. every member of the family $(G_n, \alpha_n)_{n\in\mathbb{N}}$ is a valid input for the algorithm) is efficient, if there exists a positive polynomial $P \in \mathbb{Z}[x]$ such that for every $(G_i, \alpha_i) \in (G_n, \alpha_n)_{n\in\mathbb{N}}$ the expected value of the run-time of the algorithm with input (G_i, α_i) is bounded by $P(\mathcal{S}(G_i, \alpha_i))$.

Tame and wild. An $\alpha \in \Lambda(G)$ is called "tame" or "wild", depending on how hard it is to find factorizations of elements with respect to α.

In literature, various similar definitions for "tame" and "wild" can be found.

- *Let $\alpha \in \Lambda(G)$. α is called tame, if the factorization of every $g \in G$ can be found efficiently. α is called wild, if it is not tame.*

  This definition is used for instance in [Bla09].

- *Let $\alpha \in \Lambda(G)$ with $\ell(\alpha)$ bounded by a polynomial in the degree of G (i.e. in the degree of the smallest faithful permutation representation of G). α is called tame, if the factorization of every $g \in G$ can be found in time polynomial in the degree of G. α is called wild, if it is not tame.*

  This definition is used for instance in [Mag02a].

- *Let $\alpha \in \Lambda(G)$ with $\ell(\alpha)$ bounded by a polynomial in $\log|G|$. α is called tame, if the factorization of every $g \in G$ can be found in time polynomial in $\log|G|$. α is called wild, if it is not tame.*

  This definition is used for instance in [Sva10] and [Bau12].

Weaknesses of the above definitions are the following:

- In the last two definitions, $\ell(\alpha)$ is restricted. A definition that does not impose a restriction on $\ell(\alpha)$ would be nice. "Tame" and "wild" should be defined for all logarithmic signatures.

- All of the definitions are black and white: either a logarithmic signature is tame or it is wild. However, there might be families of logarithmic signatures that are neither tame nor wild (e.g. a family of logarithmic signatures in which only a fixed percentage of elements can be factored efficiently).

  For example, let $\alpha = (A_1, \ldots, A_n) \in \Lambda(D_{2^n})$ with $t(\alpha) = (2, \ldots, 2)$, $\mathrm{id} \in A_j$ for all $1 \le j \le n$ and $\mathscr{C}_\tau(E(\alpha)) = 1$ (i.e. α contains only one reflection); let the block containing the reflection be A_i. $\alpha' := (A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n)$ generates the cyclic subgroup of all rotations in D_{2^n}, and factoring with respect to α' (as logarithmic signature of a cyclic group) is easy by Chapter 8. So, even without knowing anything about how to factor with respect to logarithmic signatures for dihedral groups in general, we already can factor 50% of the elements of D_{2^n} with respect to α. As

we will see in Proposition 9.6, factoring with respect to α is actually 100% easy; nonetheless this example shows that the case that a fixed percentage of elements can be factored easily might indeed occur in a natural way.

Intuitively, a "wild" logarithmic signature would be suited for use in a cryptographic system based on the hardness of factoring elements with respect to this logarithmic signature. However, when for instance a fixed percentage $< 100\%$ of elements is hard to factor, it would not be suitable in a cryptographic system.

A stronger definition of "wild" would realize this. The hardness of factoring a randomly chosen group element would be a good candidate.

- It is not specified how α is represented. However, this is important as we will see in Remark 3.6.

We use the definition of the terms "tame"/"wild" from [Nus11] (which does not have the weaknesses above), where the definition of "wild" relates to one-way functions. We state this definition now.

Let $(G_n, \alpha_n)_{n\in\mathbb{N}}$ be a family of groups G_n (we usually assume that $|G_n| \leq |G_{n+1}|$) and $\alpha_n \in \Lambda(G_n)$ for all $n \in \mathbb{N}$, all α_n pairwise different and encoded using a list representation. We assume that there exists a deterministic polynomial-time algorithm that computes Θ_{α_n} for all $n \in \mathbb{N}$.

- The family $(G_n, \alpha_n)_{n\in N}$ is called *tame*, if there exists a deterministic polynomial-time algorithm that computes $\Theta_{\alpha_n}^{-1}(g_n)$ for every input $g_n \in G_n$.
- The family $(G_n, \alpha_n)_{n\in N}$ is called *wild*, if for every probabilistic polynomial-time algorithm A and every positive polynomial $P \in \mathbb{Z}[x]$ the following holds for all sufficiently large $n \in \mathbb{N}$:

$$\Pr[A(G_n, \alpha_n, g_n) = \Theta_{\alpha_n}^{-1}(g_n) \mid g_n \leftarrow \Omega_{G_n}] < \frac{1}{P(\mathcal{S}(G_n, \alpha_n))},$$

where "$g_n \leftarrow \Omega_{G_n}$" means to randomly choose an element g_n from the uniform distribution on G_n.

Note that being wild is a special case of being not tame.

Notation. From now on, we usually omit the term "family of" and just call logarithmic signatures tame or wild. Instead of $(G_n, \alpha_n)_{n\in\mathbb{N}}$ we just write (G, α), keeping in mind that actually whole families are meant.

For example, by "let $\alpha \in \Lambda(G)$ with $\ell(\alpha)$ polynomial in $\log|G|$" we actually mean a family $(G_n, \alpha_n)_{n\in\mathbb{N}}$ and a polynomial $P \in \mathbb{Z}[x]$ such that $\ell(\alpha_n) \leq P(\log|G_n|)$ holds for every $n \in \mathbb{N}$.

Mapping. If $\alpha \in \Lambda(G)$ is wild, the mapping $\breve{\alpha}$ is a one-way function. Computing $\breve{\alpha}$ is possible efficiently, but computing $\breve{\alpha}^{-1}$ is hard.

One-way permutation. If $\alpha \in \Lambda(G)$ is wild and $\beta \in \Lambda(G)$ is tame, the mapping $P_{\alpha,\beta}$ is a one-way permutation on $\mathbb{Z}_{|G|}$. For every $x \in \mathbb{Z}_{|G|}$, computing $P_{\alpha,\beta}(x) = \breve{\beta}^{-1}(\breve{\alpha}(x))$ is possible efficiently (β is tame, thus $\breve{\beta}^{-1}$ can be computed efficiently). However, computing $P^{-1}_{\alpha,\beta}(x) = P_{\beta,\alpha}(x) = \breve{\alpha}^{-1}(\breve{\beta}(x))$ is hard, due to α being wild.

Group order and minimal input size. We highlight the connection between the group order and the minimal size of the input (G, α) (when α is encoded using a list representation) in the following proposition.

Proposition 3.3. *Let $(G_n, \alpha_n)_{n\in\mathbb{N}}$ be a family as above and $P \in \mathbb{Z}[x]$ a polynomial such that for every $n \in \mathbb{N}$ there exists a $p_n \in \mathbb{P}$ with $p_n \mid |G_n|$ and $|G_n| \leq P(p_n)$.*

Then $(G_n, \alpha_n)_{n\in\mathbb{N}}$ is tame.

Proof. The sizes of the blocks of α_n divide the group order $|G_n|$. Thus we have $\ell(\alpha_n) \geq p_n$ (because α_n must contain a block whose size is p_n or a multiple of p_n), i.e. $\mathcal{S}(G_n, \alpha_n) \in \Omega(p_n)$. As $|G_n|$ is polynomial in p_n, factoring elements using brute-force is possible in time polynomial in the input length. □

Example 3.4. Let $(G_n, \alpha_n)_{n\in\mathbb{N}}$ be a family with $|G_n| = p_n^2 q_n$ for $p_n, q_n \in \mathbb{P}$. This implies $\ell(\alpha_n) \geq 2p_n + q_n$. Factoring elements using brute-force requires time polynomial in $p_n^2 q_n$, and we have $p_n^2 q_n \leq (2p_n + q_n)^3$, i.e. this is polynomial in the input length.

Remark 3.5. Let $|G| = p_1^{a_1} \cdots p_k^{a_k}$ be the prime factorization of $|G|$, and $\alpha = (A_1, \ldots, A_s) \in \Lambda(G)$. Then obviously $\ell(\alpha)$ is minimal if and only if $|A_i| \in \mathbb{P} \cup \{4\}$ for all $1 \leq i \leq s$ (the size of every block must divide the group order, and a block of size $b \cdot c$ contains at least as many elements as two blocks of size b and c, because $b + c \leq b \cdot c$ holds for all $b, c \in \mathbb{N}_{\geq 2}$; removing any blocks of size 1 results in another logarithmic signature, therefore blocks of size 1 may be excluded), thus we get the inequality

$$\ell(\alpha) = \sum_{i=1}^{s} |A_i| \geq \sum_{j=1}^{k} a_j p_j.$$

So, the minimal length of a logarithmic signature is $\sum_{j=1}^{k} a_j p_j$. Minimal length logarithmic signatures are known to exist for solvable groups, symmetric and alternating groups, several classical/simple/sporadic groups and all groups of order < 175560; see [Vas03], [Hol04] and [Lem05].

A less precise lower bound clearly is $\log_2 |G| \leq \ell(\alpha)$.

So, we have

$$\log_2 |G| \leq \ell(\alpha) \leq b_G \cdot \ell(\alpha) = \mathcal{S}(G, \alpha).$$

Especially, being polynomial in $\log_2 |G|$ implies being polynomial in $\ell(\alpha)$ and in $\mathcal{S}(G, \alpha)$. If $\ell(\alpha)$ and b_G are polynomial in $\log_2 |G|$, then being polynomial in $\mathcal{S}(G, \alpha)$ is equivalent to being polynomial in $\log_2 |G|$.

Remark 3.6. Proposition 3.3 assumes that a list representation of α is used. If a different representation is used, the statement is not necessarily true anymore.

For example, let $G = \mathbb{Z}_p$ for some $p \in \mathbb{P}$ and $f : \mathbb{Z}_p \to \mathbb{Z}_p$ a one-way permutation. Assume that space polynomial in $\log|G|$ is sufficient for storing/describing f. Then

$$\alpha := (f(0), f(1), \ldots, f(p-1))$$

(one block containing p elements) is a logarithmic signature for G, and $\breve{\alpha}(x) = f(x)$ can be computed efficiently. Space polynomial in $\log|G|$ is sufficient for storing α (store the function f instead of all elements in the block explicitly). With this representation, $\breve{\alpha}^{-1}$ cannot be computed in time polynomial in $\log|G|$ (because f cannot be inverted in time polynomial in $\log|G|$).

In contrast, if α is stored using a list representation, it is tame by Proposition 3.3.

3.4. Block Sequence Classes

3.4.1. Canonical Block Sequences

A block A is called *canonical*, if $A[1] = \mathrm{id}$.

A block sequence $\alpha = (A_1, \ldots, A_s) \in \Xi_s(G)$ is called

- ℓ-*canonical*, if $A_i[1] = \mathrm{id}$ for all $i \in \{1, 2, \ldots, s-1\}$,
- r-*canonical*, if $A_i[1] = \mathrm{id}$ for all $i \in \{2, 3, \ldots, s\}$,
- *canonical*, if α is ℓ- and r-canonical,
- $\ell^\diamond$-*canonical*, if α is ℓ-canonical and $|A_s| = 1$,
- $r^\diamond$-*canonical*, if α is r-canonical and $|A_1| = 1$.

3.4.2. Periodic Block Sequences

A block sequence $\alpha = (A_1, \ldots, A_s) \in \Xi_s(G)$ is called r- or ℓ-*periodic*, if at least one block A_i is r- or ℓ-periodic.

If α is r- or ℓ-periodic, it is also called *periodic*. If none of the blocks is periodic, α is called *aperiodic*.

We define strong aperiodicity for logarithmic signatures of abelian groups like it was defined in [Sta13].

Let G be an abelian group and $\alpha = (A_1, \ldots, A_s) \in \Lambda(G)$ aperiodic. Let $1 \le i_1 < i_2 < \ldots < i_d \le s$ and α' the logarithmic signature resulting from fusing the blocks $A_{i_1}, \ldots, A_{i_d}$ in α (i.e. removing these blocks and adding one large block $A_{i_1} + \ldots + A_{i_d}$ at the end).

The fusion of the d blocks $A_{i_1}, \ldots, A_{i_d}$ is called *admissible*, if the type $t(\alpha')$ does not imply that α' is periodic (for sufficient conditions on $t(\alpha')$ such that α' is periodic, see Lemma 8.42).

Let $\{d_1, \ldots, d_z\}$ be the set of the largest possible numbers of blocks permitted by admissible fusions.

α is called *strongly aperiodic*, if for each $d \in \{d_1, \ldots, d_z\}$ every admissible fusion of d blocks results in an aperiodic logarithmic signature.

A vector $(k_1, \ldots, k_s) \in \mathbb{N}^s$ is called a *periodicity forcing factorization type for* G, if every $\alpha \in \Lambda(G)$ with $t(\alpha) = (k_1, \ldots, k_s)$ is periodic.

Various periodicity forcing factorization types for abelian groups can be found in Lemma 8.42.

3.4.3. Transversal Logarithmic Signatures

Let $\gamma : \{\mathrm{id}\} = G_0 < G_1 < \ldots < G_{s-1} < G_s = G$ be a chain of subgroups of G. Let A_i be a complete set of right coset representatives of G_{i-1} in G_i (i.e. $G_{i-1}A_i = G_i$ and $|A_i| = \frac{|G_i|}{|G_{i-1}|}$). These blocks form a logarithmic signature $\alpha := (A_1, A_2, \ldots, A_s)$ for G. Such a logarithmic signature is called *exact transversal*.

A logarithmic signature α is called *transversal*, if it is a sandwich (see Section 5.1.4) of an exact transversal logarithmic signature.

Proposition 3.7. *Let $\gamma : \{\mathrm{id}\} = G_0 < G_1 < \ldots < G_{s-1} < G_s = G$ be a chain of subgroups of G, and A_i a complete set of right coset representatives of G_{i-1} in G_i (i.e. $G_{i-1}A_i = G_i$ and $|A_i| = \frac{|G_i|}{|G_{i-1}|}$). Let $\alpha := (A_1, \ldots, A_s)$ be the corresponding exact transversal logarithmic signature for G.*

α is tame if and only if for every $g \in G$ and $0 \leq i \leq s$ it can be tested efficiently whether $g \in G_i$.

Proof. The direction "$\Rightarrow$" is clear: in order to test whether $g \in G_i$, compute the factorization of g with respect to α (tame by hypothesis) and test whether id has been selected in all blocks that are not involved in generating G_i.

We now prove "$\Leftarrow$", i.e. we compute the factorization indices $(y_1, \ldots, y_s)$ of g with respect to α (such that $g = \alpha[1][y_1] \cdots \alpha[s][y_s]$).

The first $s-1$ blocks generate the subgroup G_{s-1}. There is exactly one $a \in A_s$ such that $g \cdot a^{-1} \in G_{s-1}$. Finding this a is possible efficiently, because $|A_s| \leq \ell(\alpha)$ and group membership testing in G_{s-1} is possible efficiently by hypothesis. Let y_s be the index of a in A_s and set $g' := g \cdot a^{-1}$.

Now repeat this recursively for every block from the right to the left, i.e. search a new $a \in A_{s-1}$ such that $g' \cdot a^{-1} \in G_{s-2}$, cancel it from the end of g', and so on.

As α contains at most $\frac{\ell(\alpha)}{2}$ blocks, all in all the process requires time polynomial in $\mathcal{S}(G, \alpha)$, i.e. α is tame. □

So, factoring with respect to α is equivalent to membership testing in the subgroups G_i. However, when an arbitrary subgroup U (with $U \neq G_i$ for $0 \leq i \leq s$) is given, being able to factor in α does not help with membership testing; a subgroup chain containing U would be required.

Subgroup membership testing (and thus factoring with respect to exact transversal logarithmic signatures) is possible efficiently for instance for permutation groups [Cus00]. Algorithm 8.23 shows that exact transversal logarithmic signatures for abelian groups represented as $\mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ are tame. However, in general (e.g. for black box groups) factoring and subgroup membership testing may be hard.

Definitions.

- $\mathcal{E}(G)$ denotes the set of all exact transversal logarithmic signatures for G.
- $\mathcal{T}(G)$ denotes the set of all transversal logarithmic signatures for G.
- $\mathcal{NT}(G)$ denotes the set of all logarithmic signatures for G that are not transversal, i.e. $\mathcal{NT}(G) = \Lambda(G) \setminus \mathcal{T}(G)$.
- $\mathcal{TNT}(G)$ denotes the set of all logarithmic signatures for G where no block is a coset of a subgroup of G ("totally non-transversal").

Furthermore, for $\mathcal{F} \in \{\mathcal{E}, \mathcal{T}, \mathcal{NT}, \mathcal{TNT}\}$, define

$$\mathcal{F}^{\diamond}(G) := \{(A_1, \ldots, A_s) \in \mathcal{F}(G) \mid s \geq 2 \wedge |A_i| \geq 2 \text{ for all } 1 \leq i \leq s\}.$$

Note that for example for $G = \mathbb{Z}_p$ with $p \in \mathbb{P}$, we get $\mathcal{F}^{\diamond}(G) = \emptyset$ (due to one large block of size p).

One important observation on composing transversal logarithmic signature mappings is the following (from [Mag92]):

Proposition 3.8. *Let G be a finite non-abelian, non-hamiltonian[1] group with*

$$|G| \notin \{q, 1+q^2, 1+q^3, \frac{q^n-1}{q-1}, 2^{n-1}(2^n \pm 1), 11, 12, 15, 22, 23, 24, 28, 176, 276 \mid q = p^k \text{ for } p \in \mathbb{P}, k \in \mathbb{N}, n \in \mathbb{N}\},$$

then the set of all non-trivial transversal logarithmic signature mappings generates the whole symmetric group (by composition), i.e.

$$\langle \{\widehat{\alpha} \mid \alpha \in \mathcal{T}^{\diamond}(G)\} \rangle \cong \mathrm{Sym}(\mathbb{Z}_{|G|}).$$

Proof. See [Mag92] (Theorem 6.5). □

Later in [Car06], this was improved as follows:

[1] A hamiltonian group is a non-abelian group in which all subgroups are normal.

Proposition 3.9. *Let G be a finite non-trivial group that is not cyclic of order p or p^2 for $p \in \mathbb{P}$. Then the mappings induced by the exact transversal logarithmic signatures in $\mathcal{E}^\diamond(G)$ generate the whole symmetric group (by composition), i.e.*

$$\langle \{ \widehat{\alpha} \mid \alpha \in \mathcal{E}^\diamond(G) \} \rangle \cong \mathrm{Sym}(\mathbb{Z}_{|G|}).$$

Proof. See [Car06] (Theorem 1.2). □

Factoring. If factoring with respect to some logarithmic signatures is possible efficiently, the composite mapping induced by them can be inverted efficiently, too (by factoring with respect to the logarithmic signatures step-by-step).

Thus, a logarithmic signature can only be wild, if computing a corresponding composition of (exact) transversal logarithmic signatures and factoring with respect to these is not possible efficiently.

Generating logarithmic signatures. It is not clear whether or how Proposition 3.9 could be used for generating logarithmic signatures. Two logarithmic signatures $\alpha, \beta \in \mathcal{E}^\diamond(G)$ together induce a mapping $\widehat{\beta} \circ \widehat{\alpha}$, however it is unclear whether or how it is possible to combine α and β to another logarithmic signature $\varsigma \in \Lambda(G)$ with $\widehat{\varsigma} = \widehat{\beta} \circ \widehat{\alpha}$ (except the trivial case $|\varsigma| = 1$).

Counting exact transversal logarithmic signatures. The number of exact transversal logarithmic signatures with respect to a given chain of subgroups can be computed easily. Let $\gamma : \{\mathrm{id}\} = G_0 < G_1 < \ldots < G_{s-1} < G_s = G$ be the chain of subgroups of G, and $\alpha = (A_1, A_2, \ldots, A_s)$ an exact transversal logarithmic signature for it. Let r_i be the number of cosets of G_{i-1} in G_i, i.e. $r_i = \frac{|G_i|}{|G_{i-1}|}$; so $(r_1, r_2, \ldots, r_s)$ is the type of α. In one coset, there are $\frac{|G_i|}{|A_i|} = \frac{|G_i|}{r_i} = \frac{|G_i|}{\frac{|G_i|}{|G_{i-1}|}} = |G_{i-1}|$ elements. These representatives of course can freely be interchanged. Consequently for a block A_i there are $|G_{i-1}|^{r_i}$ possibilities for the coset representatives. Finally, we can permute the order of the representatives in each block, resulting in $|G_{i-1}|^{r_i} \cdot r_i!$ possibilities for block A_i.

So, in total the number of exact transversal logarithmic signatures with respect to the above chain of subgroups is

$$\prod_{i=1}^{s} (|G_{i-1}|^{r_i} \cdot r_i!) = \prod_{i=1}^{s} \left(\left(\prod_{j=1}^{i-1} r_j \right)^{r_i} \cdot r_i! \right).$$

Note that some of these exact transversal logarithmic signatures may be equivalent, i.e. there exist exact transversal $\alpha, \beta \in \Lambda(G)$ with respect to the same subgroup chain with $\alpha \neq \beta$, but $\breve{\alpha} = \breve{\beta}$.

For example, let $G = \mathbb{Z}_4$, $\alpha = \binom{0}{2}\binom{0}{1}$ and $\beta = \binom{2}{0}\binom{2}{3}$ both exact transversal with respect to the subgroup chain $\{0\} < \{0, 2\} < G$, $\alpha \neq \beta$. However, we have $\breve{\alpha} = \breve{\beta}$, due to β being a sandwich of α:

$$\alpha = \binom{0}{2}\binom{0}{1} \breve{=} \binom{0}{2}(2)(2)\binom{0}{1} \breve{=} \binom{2}{0}\binom{2}{3} = \beta.$$

By Proposition 5.6 we know that two logarithmic signatures of the same type are equivalent if and only if they are sandwiches of each other.

Not all sandwiches of an exact transversal logarithmic signature necessarily are again exact transversal though. For example, $\binom{0}{2}(1)(3)\binom{0}{1} \stackrel{\sim}{=} \binom{1}{3}\binom{3}{0}$ is a sandwich of α (and thus again a logarithmic signature for G), but it is not exact transversal (e.g. because the first block is not a subgroup).

In order to count the inequivalent exact transversal logarithmic signatures, we divide the number above by the number of possible sandwiches (possible in the sense that the sandwich is again exact transversal). For the first sandwich element (between the blocks A_1 and A_2), there are $|A_1|$ possible sandwich elements, namely the elements of A_1, because A_1 must remain a subgroup. For the second sandwich element (between the blocks A_2 and A_3), there are $|A_1| \cdot |A_2|$ possible sandwich elements, namely the elements of $A_1 \cdot A_2$. And so on.

Thus, the number of inequivalent exact transversal logarithmic signatures is

$$\begin{aligned}
\frac{\prod_{i=1}^{s}\left(|G_{i-1}|^{r_i}\cdot r_i!\right)}{\prod_{i=1}^{s-1}\prod_{j=1}^{i} r_j} &= \frac{\prod_{i=1}^{s}\left(\left(\prod_{j=1}^{i-1} r_j\right)^{r_i}\cdot r_i!\right)}{\prod_{i=1}^{s-1}\prod_{j=1}^{i} r_j} \\
&= \prod_{i=1}^{s}\frac{\left(\prod_{j=1}^{i-1} r_j\right)^{r_i}\cdot r_i!}{\prod_{j=1}^{i} r_j}\cdot\prod_{j=1}^{s} r_j \\
&= \prod_{i=1}^{s}\left(\left(\prod_{j=1}^{i-1} r_j\right)^{r_i-1}\cdot(r_i-1)!\right)\cdot\prod_{j=1}^{s} r_j \\
&= \prod_{i=1}^{s}\left(\left(\prod_{j=1}^{i-1} r_j\right)^{r_i-1}\cdot r_i!\right).
\end{aligned}$$

4. Cryptographic Primitives

In Chapter 4, we give an overview on existing cryptographic primitives based on group factorizations.

4.1. PGM (Symmetric Encryption, Logarithmic Signature)

Permutation Group Mappings (PGM) is a symmetric encryption algorithm that uses logarithmic signatures as keys. It has been introduced by S. S. Magliveras [Mag86] [Mag92].

Let G be a group.

- **Spaces.** Message space: $\mathbb{Z}_{|G|}$, cipher space: $\mathbb{Z}_{|G|}$.
- **Key.** $(\alpha, \beta) \in \Lambda(G)^2$.
- **Encryption.**
$$E_{\alpha,\beta} : \mathbb{Z}_{|G|} \to \mathbb{Z}_{|G|} : x \mapsto P_{\alpha,\beta}(x) = \breve{\beta}^{-1}(\breve{\alpha}(x)).$$
- **Decryption.**
$$D_{\alpha,\beta} : \mathbb{Z}_{|G|} \to \mathbb{Z}_{|G|} : x \mapsto P^{-1}_{\alpha,\beta}(x) = \breve{\alpha}^{-1}(\breve{\beta}(x)).$$

Note that for encryption β must be tame, and for decryption α must be tame, i.e. both must be tame.

Security. When performing multiple encryptions, PGM has a strong cryptographic property: almost always all permutations of $\mathbb{Z}_{|G|}$ can be achieved, due to Proposition 3.8. With one encryption, PGM is usually 2-transitive [Mag92].

In [Mag92], a computationally infeasible chosen-plaintext attack is described.

4.2. MST_1 (Asymmetric Encryption, Logarithmic Signature)

MST_1 is an asymmetric encryption algorithm that uses logarithmic signatures. It has been introduced by S. S. Magliveras, D. R. Stinson and T. van Trung [Mag02b].

Let G be a group.

- **Spaces.** Message space: $\mathbb{Z}_{|G|}$, cipher space: $\mathbb{Z}_{|G|}$.

- **Public key.** $(\alpha, \beta) \in \Lambda(G)^2$ with α wild (without the knowledge of the private key) and β tame.
- **Private key.** Tame transversal logarithmic signatures $\theta_1, \ldots, \theta_k \in \Lambda(G)$ such that $\widehat{\alpha}\widehat{\beta}^{-1} = \widehat{\theta_1} \cdots \widehat{\theta_k}$ (such a composition usually exists by Proposition 3.8).
- **Encryption.**
$$E_{\alpha,\beta} : \mathbb{Z}_{|G|} \to \mathbb{Z}_{|G|} : x \mapsto P_{\alpha,\beta}(x) = \breve{\beta}^{-1}(\breve{\alpha}(x)).$$
- **Decryption.**
$$D_{\alpha,\beta} : \mathbb{Z}_{|G|} \to \mathbb{Z}_{|G|} : x \mapsto P_{\alpha,\beta}^{-1}(x) = \breve{\alpha}^{-1}(\breve{\beta}(x)).$$

Security. The security of this system relies on the cryptographic hypothesis that factoring with respect to logarithmic signatures in general is an intractable problem. For a detailed analysis, see [Mag02b].

4.3. MST_1 Generalized (Asymmetric Encryption, Logarithmic Signature)

In the definition of MST_1 above, $\widehat{\alpha}\widehat{\beta}^{-1}$ must be expressible by a composition of mappings induced by tame transversal logarithmic signatures. This is motivated by the fact that usually the set of all mappings induced by (exact) transversal logarithmic signatures generates the whole symmetric group $\mathrm{Sym}(\mathbb{Z}_{|G|})$ (when composing the mappings induced by the logarithmic signatures with a fixed tame logarithmic signature like β), see the Propositions 3.8 and 3.9. So, even wild logarithmic signatures can usually be written as a composition of transversals; the wildness relies on not knowing the transversals (without the private key).

However, more generally it would be sufficient when the knowledge of the private key allows efficient factoring with respect to α, i.e. α is tame when knowing the private key and it is wild when not knowing it.

The format of the private key is dependent on the algorithm that allows efficient factoring with respect to the public logarithmic signature.

So, we obtain the following generalized definition of MST_1.

Let G be a group.

- **Spaces.** Message space: $\mathbb{Z}_{|G|}$, cipher space: $\mathbb{Z}_{|G|}$.
- **Public key.** $(\alpha, \beta) \in \Lambda(G)^2$ with α wild (without the knowledge of the private key) and β tame.
- **Private key.** Information with which α is tame.
- **Encryption.**
$$E_{\alpha,\beta} : \mathbb{Z}_{|G|} \to \mathbb{Z}_{|G|} : x \mapsto P_{\alpha,\beta}(x) = \breve{\beta}^{-1}(\breve{\alpha}(x)).$$

- **Decryption.**
$$D_{\alpha,\beta} : \mathbb{Z}_{|G|} \to \mathbb{Z}_{|G|} : x \mapsto P^{-1}_{\alpha,\beta}(x) = \breve{\alpha}^{-1}(\breve{\beta}(x)).$$

Security. See Section 4.2.

4.4. MST_2 (Asymmetric Encryption, $[s, r]$-Mesh)

MST_2 is an asymmetric encryption algorithm that uses $[s, r]$-meshes. It has been introduced by S. S. Magliveras, D. R. Stinson and T. van Trung [Mag02b].

Let G, H be groups such that there exist surjective homomorphisms from G to H.

- **Spaces.** Message space: H, cipher space: $G \times H$.
- **Private key.** Surjective homomorphism $f : G \to H$.
- **Public key.** (α, β), where α is a random $[s, r]$-mesh for G and $\beta := f(\alpha)$.
- **Encryption.**
  1. Choose a random $z \in \mathbb{Z}_{r^s}$.
  2. Compute $y_1 := \breve{\alpha}(z)$, $y_2 := \breve{\beta}(z)$, $y_3 := hy_2$, where $h \in H$ is the message to be sent.
  3. Send (y_1, y_3).
- **Decryption.**
  1. Recover y_2 by computing $f(y_1) = f(\breve{\alpha}(z)) = \breve{\beta}(z) = y_2$.
  2. Obtain the message h by computing $y_3 y_2^{-1} = h$.

Security. The security of this system relies on the cryptographic hypothesis that factoring with respect to $[s, r]$-meshes in general is an intractable problem. Furthermore, without knowing the private key it must be computationally infeasible to find a homomorphism $f' : G \to H$ with $\beta = f'(\alpha)$. For a detailed analysis, see [Mag02b].

For cyclic groups, factoring with respect to an $[s, r]$-mesh is at least as hard as the discrete logarithm problem [Cus00].

4.5. MST_3 (Asymmetric Encryption, Cover)

MST_3 is an asymmetric encryption algorithm that uses covers and logarithmic signatures. It has been introduced by W. Lempken, T. van Trung, S. S. Magliveras and W. Wei [Lem09].

Let G be a non-abelian group with a large center $\mathcal{Z}$ such that G does not split over $\mathcal{Z}$ (i.e. there exists no $H \leq G$ with $G = \mathcal{Z} \times H$). $|\mathcal{Z}|$ should be large enough such that exhaustive searching in $\mathcal{Z}$ is computationally infeasible.

- **Spaces.** Message space: $\mathbb{Z}_{|\mathcal{Z}|}$, cipher space: G^2.
- **Private setup.**
  1. Generate a tame $\beta = (B_1, \ldots, B_s) \in \Lambda(\mathcal{Z})$.
  2. Generate a random cover $\alpha = (A_1, \ldots, A_s)$ with $t(\alpha) = t(\beta)$ for a subset of G such that $A_1, \ldots, A_s \subseteq G \setminus \mathcal{Z}$.
  3. Choose random $t_0, \ldots, t_s \in G \setminus \mathcal{Z}$.
  4. Compute $\widetilde{\alpha} := (\widetilde{A}_1, \ldots, \widetilde{A}_s)$ with $\widetilde{A}_i := t_{i-1}^{-1} \cdot A_i \cdot t_i$ for $1 \leq i \leq s$.
  5. Compute $\gamma := (H_1, \ldots, H_s)$ with $|H_i| = |B_i|$, $H_i[j] = B_i[j] \cdot \widetilde{A}_i[j]$ for $1 \leq i \leq s$, $1 \leq j \leq |B_i|$.
- **Private key.** $(\beta, (t_0, \ldots, t_s))$.
- **Public key.** (α, γ).
- **Encryption.** Let $m \in \mathbb{Z}_{|\mathcal{Z}|}$ be the message to be sent.
  1. Compute $y_1 := \breve{\alpha}(m)$ and $y_2 := \breve{\gamma}(m)$.
  2. Send (y_1, y_2).
- **Decryption.** Let $(y_1, y_2) \in G^2$ be the encrypted message.
  1. Compute $y_2 \cdot t_s^{-1} \cdot y_1^{-1} \cdot t_0 = \breve{\beta}(m)$.
  2. Recover m from $\breve{\beta}(m)$ ($\breve{\beta}^{-1}$ can be computed efficiently, because β is tame).

Correctness. The decryption works, because

$$\begin{aligned} y_2 = \breve{\gamma}(m) &= B_1[j_1] \cdot \widetilde{A}_1[j_1] \cdots B_s[j_s] \cdot \widetilde{A}_s[j_s] \\ &= B_1[j_1] \cdot t_0^{-1} \cdot A_1[j_1] \cdot t_1 \cdots B_s[j_s] \cdot t_{s-1}^{-1} \cdot A_s[j_s] \cdot t_s \\ &= B_1[j_1] \cdots B_s[j_s] \cdot t_0^{-1} \cdot A_1[j_1] \cdots A_s[j_s] \cdot t_s = \breve{\beta}(m) \cdot t_0^{-1} \cdot \breve{\alpha}(m) \cdot t_s \\ &= \breve{\beta}(m) \cdot t_0^{-1} \cdot y_1 \cdot t_s. \end{aligned}$$

Security. The security of this system relies on the cryptographic hypothesis that factoring with respect to covers for large subsets of groups in general is an intractable problem. For a detailed analysis, see [Lem09].

4.6. MST$_g$ (Pseudo-Random Number Generator, Cover)

MST$_g$ is a pseudo-random number generator (PRNG) that uses random covers. There is evidence that it is cryptographically secure (in the sense of a CSPRNG as defined in Section 2.7). MST$_g$ has been introduced by P. Marquardt, P. Svaba and T. van Trung [Mar12].

Let G_1, G_2 be groups with $|G_1| \geq |G_2|$, and $f_1 : G_1 \to \mathbb{Z}_{|G_1|}$, $f_2 : G_2 \to \mathbb{Z}_{|G_2|}$ bijective mappings. Let $l \in \mathbb{N}$ with $l \geq |G_1|$, and $k \in \mathbb{N}_0$.

- **Instance generation.** Generate:
  - A random cover α for G_1 with $t(\alpha) = (u_1, \ldots, u_t)$ such that $\prod_{i=1}^{t} u_i = l$.
  - Random covers $\alpha_1, \ldots, \alpha_k$ for G_1 with $t(\alpha_i) = (r_1, \ldots, r_s)$, $\prod_{i=1}^{s} r_i = |G_1|$ for all $1 \leq i \leq k$.
  - A random cover $\gamma = (H_1, \ldots, H_s)$ for G_2 with $t(\gamma) = (r_1, \ldots, r_s)$.

  Define a mapping $F : \mathbb{Z}_l \to \mathbb{Z}_{|G_2|}$ by the following composition:

$$\mathbb{Z}_l \xrightarrow{\breve{\alpha}} G_1 \xrightarrow{f_1} \mathbb{Z}_{|G_1|} \xrightarrow{\breve{\alpha}_1} G_1 \xrightarrow{f_1} \mathbb{Z}_{|G_1|} \longrightarrow \ldots \xrightarrow{\breve{\alpha}_k} G_1 \xrightarrow{f_1} \mathbb{Z}_{|G_1|} \xrightarrow{\breve{\gamma}} G_2 \xrightarrow{f_2} \mathbb{Z}_{|G_2|}.$$

- **Input.** Seed $s_0 \in \mathbb{Z}_l$, constant $C \in \mathbb{Z}_l$ (with $\gcd(C, l) = 1$).
- **Output.** To generate n pseudo-random numbers $z_1, \ldots, z_n \in \mathbb{Z}_{|G_2|}$, do the following:
  - For $i \leftarrow 1$ to n:
    1. Set $s_i \leftarrow (s_{i-1} + C) \bmod l$.
    2. Output $z_i := F(s_i)$.

Analysis. MST_g can be highly efficient, and evidence of excellent statistical and cryptographic properties has been shown [Mar12].

5. Transformations and Irreducibility

In Chapter 5, various transformations on block sequences are presented. Furthermore, we analyze the irreducibility of blocks and its connection to linear representations and characters.

Transformations and irreducibility conditions are interesting, because they can be used in block sequence generation and factorization algorithms.

Our contributions. Our first main contribution in this chapter is a rigorous analysis of transformations on block sequences. We analyze their effect on factorization mappings, define subclasses of transformations (e.g. factorization-permuting block shuffles, which are interesting for block sequence generation algorithms that include block shuffle transformations). New, interesting types of block sequence normalizations (which we call g- and (i, g)-normalizations) are presented. Block substitutions are introduced, and we have a look at their efficiency and a possible implementation. We analyze compositions of transformations in detail (e.g. which compositions form a group); this is especially important for designing block sequence generation algorithms based on iterated transformations. In light of this, we emphasize one special combination (translation, element shuffle and normalization), which we call a TSN transformation and which plays a significant role in our logarithmic signature generation algorithm presented in Chapter 6.

Another contribution in this chapter is an analysis of irreducibility of blocks, mainly based on group homomorphisms. Linear representations and characters are useful for studying factorizations of abelian groups; we contribute an analysis that shows to what extent some results for abelian groups can be generalized for non-abelian groups.

5.1. Transformations

In the following we have a look at several transformations that can be applied to block sequences.

We are especially interested in transformations that produce new, valid logarithmic signatures when the input is a logarithmic signature.

5.1.1. Element Shuffle

The blocks $A_1, A_2, \ldots, A_s$ of a block sequence are sequences of elements of G. It is clear that permuting the elements within a block A_i does not change the multiset of elements generated by the block sequence.

We call permuting the elements within blocks an *element shuffle*.

Shuffling elements within blocks of a k-factorization results in another k-factorization.

Let

$$\mathfrak{e}_{(\pi_1,\ldots,\pi_s)} : \Xi_{(r_1,\ldots,r_s)}(G) \to \Xi_{(r_1,\ldots,r_s)}(G) : (A_1,\ldots,A_s) \mapsto \begin{pmatrix} A_1[\pi_1(1)] \\ \vdots \\ A_1[\pi_1(r_1)]) \end{pmatrix} \cdots \begin{pmatrix} A_s[\pi_s(1)] \\ \vdots \\ A_s[\pi_s(r_s)] \end{pmatrix}$$

be the transformation that applies an element shuffle using the permutations $\pi_1 \in \mathrm{Sym}(r_1),\ldots,\pi_s \in \mathrm{Sym}(r_s)$ to a block sequence of type $(r_1,\ldots,r_s)$. Let

$$\mathfrak{E}_{(r_1,\ldots,r_s)}(G) := \{\mathfrak{e}_{(\pi_1,\ldots,\pi_s)} \mid \pi_1 \in \mathrm{Sym}(r_1),\ldots,\pi_s \in \mathrm{Sym}(r_s)\}$$

be the set of all element shuffle transformations.

Together with composition as binary operation, $\mathfrak{E}_{(r_1,\ldots,r_s)}(G)$ is a group. We have $\mathfrak{e}_{(\widetilde{\pi}_1,\ldots,\widetilde{\pi}_s)} \circ \mathfrak{e}_{(\pi_1,\ldots,\pi_s)} = \mathfrak{e}_{(\widetilde{\pi}_1\circ\pi_1,\ldots,\widetilde{\pi}_s\circ\pi_s)}$, the identity is $\mathfrak{e}_{(\mathrm{id},\ldots,\mathrm{id})}$, and $\mathfrak{e}^{-1}_{(\pi_1,\ldots,\pi_s)} = \mathfrak{e}_{(\pi_1^{-1},\ldots,\pi_s^{-1})}$.

Let $r_{\max} := \max\{r_1,\ldots,r_s\}$, then $\mathfrak{E}_{(r_1,\ldots,r_s)}(G)$ is non-abelian if and only if $r_{\max} \geq 3$ (because $\mathrm{Sym}(n)$ is non-abelian if and only if $n \geq 3$).

We would like to highlight the effect of an element shuffle on the mapping $\breve{\alpha}$ induced by an $\alpha \in \Xi_s(G)$ of type $t(\alpha) = (r_1,\ldots,r_s)$. Let $\pi_1 \in \mathrm{Sym}(r_1),\ldots,\pi_s \in \mathrm{Sym}(r_s)$. Recall that the function $\breve{\alpha}$ is defined (in Section 3.2) as $\breve{\alpha} : \mathbb{Z}_{\prod_{i=1}^s r_i} \to G : x \mapsto \Theta_\alpha(\lambda^{-1}(x))$, where $\lambda : \mathbb{Z}_{r_1} \oplus \mathbb{Z}_{r_2} \oplus \ldots \oplus \mathbb{Z}_{r_s} \to \mathbb{Z}_{\prod_{i=1}^s r_i} : (i_1,i_2,\ldots,i_s) \mapsto \sum_{j=1}^s i_j m_j$ with $m_j := \prod_{k=1}^{j-1} r_k$. Let $\beta := \mathfrak{e}_{(\pi_1,\ldots,\pi_s)}(\alpha)$ and $\widetilde{\lambda} : \mathbb{Z}_{r_1} \oplus \mathbb{Z}_{r_2} \oplus \ldots \oplus \mathbb{Z}_{r_s} \to \mathbb{Z}_{\prod_{i=1}^s r_i} : (i_1,i_2,\ldots,i_s) \mapsto \sum_{j=1}^s ((\pi_j(i_j+1)-1)\cdot m_j)$, we then get

$$\begin{aligned} \breve{\beta} : \mathbb{Z}_{|G|} \to G : x &\mapsto \Theta_\beta(\lambda^{-1}(x)) \\ &= \Theta_\alpha(\widetilde{\lambda}^{-1}(x)). \end{aligned}$$

Note that $(\pi_1,\ldots,\pi_s) \neq (\mathrm{id},\ldots,\mathrm{id})$ is not a sufficient condition for $\breve{\beta} \neq \breve{\alpha}$. For example, let $G = \mathbb{Z}_2$, $\alpha = ((0,1),(0,1)) \in \Xi(G)$ (not a logarithmic signature), $\pi_1 = \pi_2 = (1\ 2) \in \mathrm{Sym}(2)$ (in cycle notation), then $\beta = \mathfrak{e}_{(\pi_1,\ldots,\pi_s)}(\alpha) = ((1,0),(1,0))$, and

$$\beta = \binom{1}{0}\binom{1}{0} \breve{=} \binom{1}{0}(1)(1)\binom{1}{0} \breve{=} \binom{0}{1}\binom{0}{1} = \alpha,$$

i.e. $\breve{\beta} = \breve{\alpha}$.

However, when α is a logarithmic signature, the condition is necessary and sufficient, as Proposition 5.1 shows.

Proposition 5.1. *Let $\alpha \in \Lambda(G)$ of type $t(\alpha) = (r_1,\ldots,r_s)$ and $\beta := \mathfrak{e}_{(\pi_1,\ldots,\pi_s)}(\alpha)$ with $\pi_1 \in \mathrm{Sym}(r_1),\ldots,\pi_s \in \mathrm{Sym}(r_s)$.*

$\breve{\beta} = \breve{\alpha}$ holds if and only if $(\pi_1,\ldots,\pi_s) = (\mathrm{id},\ldots,\mathrm{id})$.

Proof. The direction "⇐" is clear: if $(\pi_1, \ldots, \pi_s) = (\mathrm{id}, \ldots, \mathrm{id})$, we get $\beta = \alpha$ and thus $\breve{\beta} = \breve{\alpha}$.

We now prove "⇒". Let $g \in G$. As α is a logarithmic signature, there exists exactly one factorization $g = a_1 \cdots a_s$ with $a_1 \in \alpha[1], \ldots, a_s \in \alpha[s]$. An element shuffle just permutes elements within blocks, thus $a_1 \cdots a_s$ also is the factorization of g with respect to β. Let $(i_1, \ldots, i_s)$ be the factorization index vector of g with respect to α, and $(j_1, \ldots, j_s)$ the factorization index vector of g with respect to β; with this, we have $\alpha[1][i_1] = a_1 = \beta[1][j_1], \ldots, \alpha[s][i_s] = a_s = \beta[s][j_s]$. $\alpha \in \Lambda(G)$ also implies that $\breve{\alpha}$ and $\breve{\beta}$ are bijective. Thus $\breve{\beta}^{-1}(g) = \breve{\alpha}^{-1}(g) \Leftrightarrow (j_1, \ldots, j_s) = (i_1, \ldots, i_s) \Leftrightarrow \pi_1(i_1) = i_1, \ldots, \pi_s(i_s) = i_s$. This holds for all $g \in G$ at once (i.e. $\breve{\beta}^{-1} = \breve{\alpha}^{-1}$) if and only if $(\pi_1, \ldots, \pi_s) = (\mathrm{id}, \ldots, \mathrm{id})$. □

5.1.2. Block Shuffle

We call permuting the order of blocks a *block shuffle*.

A block shuffle of a logarithmic signature for a group G can result in another valid logarithmic signature for G or only in a pseudo-logarithmic signature for G. For abelian groups G, the result always is a logarithmic signature. For non-abelian groups, both can happen.

Example 5.2. Let $G = D_{2\cdot 8}$, the dihedral group of order 16.

- $\alpha = ((\mathrm{id}, \tau), (\mathrm{id}, \sigma^6), (\mathrm{id}, \sigma), (\mathrm{id}, \sigma^5\tau))$ is a logarithmic signature for G.
- $\alpha = ((\mathrm{id}, \tau), (\mathrm{id}, \sigma), (\mathrm{id}, \sigma^6), (\mathrm{id}, \sigma^5\tau))$ is another logarithmic signature for G (obviously, as the two swapped blocks only contain rotations, i.e. are elementwise permutable).
- $\alpha = ((\mathrm{id}, \sigma^5\tau), (\mathrm{id}, \sigma^6), (\mathrm{id}, \sigma), (\mathrm{id}, \tau))$ is only a pseudo-logarithmic signature. For example, the element $\sigma^7\tau \in G$ has two factorizations: $\mathrm{id} \cdot \sigma^6 \cdot \sigma \cdot \tau = \sigma^7\tau$ and $\sigma^5\tau \cdot \sigma^6 \cdot \mathrm{id} \cdot \mathrm{id} = \sigma^{5-6 \bmod 8}\tau = \sigma^7\tau$.

Definition 5.3. Let $A_1, \ldots, A_n \subseteq G$, $A = A_1 \cdots A_n$ and $\pi \in \mathrm{Sym}(n)$. Assume that $A_{\pi(1)}, \ldots, A_{\pi(n)}$ is a valid block shuffle (i.e. $A_{\pi(1)} \cdots A_{\pi(n)} = A$).

The block shuffle is called *factorization-permuting*, if $A_{\pi(1)}[i_{\pi(1)}] \cdots A_{\pi(n)}[i_{\pi(n)}] = g$ for every $g \in A$ with factorization indices $i_1, \ldots i_n$ in the non-permuted block sequence (i.e. $g = A_1[i_1] \cdots A_n[i_n]$).

Example 5.4. If G is abelian, all block shuffles are factorization-permuting.

Example 5.5. Let $G = D_{2\cdot 8}$ and $\alpha = \binom{\mathrm{id}}{\sigma}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\tau}\binom{\mathrm{id}}{\sigma^4} \in \Lambda(G)$. Observe that all block shuffles are valid, i.e. every permutation of the blocks of α results in another logarithmic signature for G.

The block shuffle $\alpha' := \binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma}\binom{\mathrm{id}}{\tau}\binom{\mathrm{id}}{\sigma^4}$ (based on the permutation $\pi = (1\ 2) \in \mathrm{Sym}(4)$) is factorization-permuting: let $g = A_1[i_1] \cdots A_4[i_4]$ with respect to α, then in order to get the factorization of g with respect to α' one needs to select the $i_{\pi(1)}$th element in the first block of α', the $i_{\pi(2)}$th element in the second block of α', etc.

In contrast, the block shuffle $\alpha'' := \binom{\mathrm{id}}{\sigma}\binom{\mathrm{id}}{\tau}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma^4}$ (based on the permutation $\pi = (2\ 3) \in \mathrm{Sym}(4)$) is not factorization-permuting. For example, let $g = \sigma^2\tau$. The factorization of g with respect to α is $\binom{\boxed{\mathrm{id}}}{\sigma}\binom{\mathrm{id}}{\boxed{\sigma^2}}\binom{\mathrm{id}}{\boxed{\tau}}\binom{\boxed{\mathrm{id}}}{\sigma^4}$. With respect to α'' the factorization is $\binom{\boxed{\mathrm{id}}}{\sigma}\binom{\mathrm{id}}{\boxed{\tau}}\binom{\mathrm{id}}{\boxed{\sigma^2}}\binom{\mathrm{id}}{\boxed{\sigma^4}}$. Observe that even though the last block A_4 was not moved by the block shuffle, different elements need to be selected in A_4 in α and α'' for the factorization of g.

5.1.3. Translation

In order to get a *left translation* of a block sequence $\alpha = (A_1, A_2, \ldots, A_s)$ by an element $t \in G$, we multiply all elements in the first block A_1 by t from the left. For a *right translation*, all elements in the last block A_s are multiplied by t from the right.

As multiplying by a group element $t \in G$ is a bijection, i.e. $tG = G$ and $Gt = G$, a translation of a logarithmic signature always is a logarithmic signature.

Finding factorizations in translated logarithmic signatures is equivalently hard as in the original logarithmic signatures. For example, if we can factorize in αt (right translation of α by $t \in G$), we can do the same in α: factoring an element $x \in G$ in α is equivalent to factoring xt in αt.

Let $\mathfrak{t}_{x,y} : \Xi(G) \to \Xi(G) : \alpha \mapsto x\alpha y$ be the transformation that applies a left translation by $x \in G$ and a right translation by $y \in G$ to a block sequence α. Let $\mathfrak{T}(G) := \{\mathfrak{t}_{x,y} \mid x, y \in G\}$ be the set of all translation transformations.

Together with composition as binary operation, $\mathfrak{T}(G)$ is a group. We have $\mathfrak{t}_{x_2,y_2} \circ \mathfrak{t}_{x_1,y_1} = \mathfrak{t}_{x_2x_1,y_1y_2}$, the identity is $\mathfrak{t}_{\mathrm{id},\mathrm{id}}$, and $\mathfrak{t}_{x,y}^{-1} = \mathfrak{t}_{x^{-1},y^{-1}}$.

$\mathfrak{T}(G)$ is abelian if and only if G is abelian.

5.1.4. Sandwich

We define a transformation $\Xi_{(r_1,\ldots,r_s)}(G) \times G^{s+1} \to \Xi_{(r_1,\ldots,r_s)}(G)$:

$$(\alpha, t) \mapsto \alpha^t = \begin{pmatrix} t_1^{-1} \cdot a_{1,1} \cdot t_2 \\ t_1^{-1} \cdot a_{1,2} \cdot t_2 \\ \vdots \\ t_1^{-1} \cdot a_{1,r_1} \cdot t_2 \end{pmatrix} \begin{pmatrix} t_2^{-1} \cdot a_{2,1} \cdot t_3 \\ t_2^{-1} \cdot a_{2,2} \cdot t_3 \\ \vdots \\ t_2^{-1} \cdot a_{2,r_2} \cdot t_3 \end{pmatrix} \cdots \begin{pmatrix} t_s^{-1} \cdot a_{s,1} \cdot t_{s+1} \\ t_s^{-1} \cdot a_{s,2} \cdot t_{s+1} \\ \vdots \\ t_s^{-1} \cdot a_{s,r_s} \cdot t_{s+1} \end{pmatrix}.$$

If $t_1 = t_{s+1} = \mathrm{id}$, α^t is called a *sandwich* of α. When building a product in a sandwich α^t, the elements inserted by t cancel out each other ($t_i \cdot t_i^{-1} = \mathrm{id}$), i.e. a factorization (vector of indices) of an element $x \in G$ in α^t is exactly the same as in α.

A left translation of an $\alpha \in \Xi(G)$ from Section 5.1.3 is nothing else than α^t with $t_2 = t_3 = \ldots = t_{s+1} = \mathrm{id}$. A right translation is α^t with $t_1 = t_2 = \ldots = t_s = \mathrm{id}$.

For all $\alpha \in \Lambda(G)$, we have $\alpha^t \in \Lambda(G)$.

Proposition 5.6. *Let $\alpha, \beta \in \Lambda(G)$ with $t(\alpha) = t(\beta)$. Then $\breve{\alpha} = \breve{\beta}$ holds if and only if α is a sandwich of β.*

Proof. See [Mag92]. □

Let

$$\mathfrak{s}_{(x_1,\ldots,x_{s-1})} : \Xi_s(G) \to \Xi_s(G) : \alpha = (A_1, \ldots, A_s) \mapsto (A_1 x_1, x_1^{-1} A_2 x_2, \ldots, x_{s-1}^{-1} A_s)$$

be the transformation that applies a sandwich using the tuple $(x_1, \ldots, x_{s-1}) \in G^{s-1}$ to the block sequence α; we have $\mathfrak{s}_{(x_1,\ldots,x_{s-1})}(\alpha) = \alpha^t$ with $t = (\mathrm{id}, x_1, \ldots, x_{s-1}, \mathrm{id})$. Let

$$\mathfrak{S}_s(G) := \{\mathfrak{s}_{(x_1,\ldots,x_{s-1})} \mid x_1, \ldots, x_{s-1} \in G\}$$

be the set of all sandwich transformations.

Together with composition as binary operation, $\mathfrak{S}_s(G)$ is a group. We have $\mathfrak{s}_{(y_1,\ldots,y_{s-1})} \circ \mathfrak{s}_{(x_1,\ldots,x_{s-1})} = \mathfrak{s}_{(x_1 y_1,\ldots,x_{s-1} y_{s-1})}$, the identity is $\mathfrak{s}_{(\mathrm{id},\ldots,\mathrm{id})}$, and $\mathfrak{s}^{-1}_{(x_1,\ldots,x_{s-1})} = \mathfrak{s}_{(x_1^{-1},\ldots,x_{s-1}^{-1})}$.

For $s \geq 2$, $\mathfrak{S}_s(G)$ is abelian if and only if G is abelian.

5.1.5. Normalization

For every $\alpha = (A_1, A_2, \ldots, A_s) \in \Xi_s(G)$ there exists a sandwich α^t with $\alpha^t[i][1] = \mathrm{id}$ for all $1 \leq i \leq s-1$, i.e. the sandwich α^t is an ℓ-canonical block sequence. Precisely, this is α^t with

$$\begin{aligned} t_1 &= \mathrm{id}, \\ t_2 &= \alpha[1][1]^{-1}, \\ t_3 &= \alpha[2][1]^{-1} \cdot \alpha[1][1]^{-1}, \\ &\vdots \\ t_s &= \alpha[s-1][1]^{-1} \cdots \alpha[1][1]^{-1}, \\ t_{s+1} &= \mathrm{id}\,. \end{aligned}$$

We call the process of computing this sandwich a *normalization*.

Algorithm 5.7. The following algorithm computes the ℓ-canonical equivalent of a block sequence $\alpha = (A_1, A_2, \ldots, A_s) \in \Xi_s(G)$:

Function `Normalize`(BlockSequence α) : Void

- For $i \leftarrow 1$ to $s-1$:
  1. Let $g \leftarrow \alpha[i][1]$.

2. For $j \leftarrow 1$ to r_i: set $\alpha[i][j] \leftarrow \alpha[i][j] \cdot g^{-1}$.
3. For $j \leftarrow 1$ to r_{i+1}: set $\alpha[i+1][j] \leftarrow g \cdot \alpha[i+1][j]$.

Proposition 5.8. *Algorithm 5.7 performs less than $2 \cdot \ell(\alpha)$ group element multiplications (which are the most expensive operations here).*

Proof. In each of the $s-1$ outer iterations, one group element multiplication is performed for each element in the block A_i and for each element in the block A_{i+1}. In total, we perform

$$\sum_{i=1}^{s-1}(|A_i| + |A_{i+1}|) = |A_1| + 2 \cdot \sum_{i=2}^{s-1} |A_i| + |A_s| < 2 \cdot \sum_{i=1}^{s} |A_i| = 2 \cdot \ell(\alpha)$$

group element multiplications. □

5.1.6. g- and (i, g)-Normalizations

Similar to the normalization in Section 5.1.5, a *g-normalization* is the process of computing a sandwich, however with different sandwich elements.

Proposition 5.9. *Let $\alpha = (A_1, A_2, \ldots, A_s) \in \Xi_s(G)$ and $g \in A_1 \cdots A_s$ arbitrary.*

There exists a sandwich of α such that in the resulting block sequence $\beta = (A'_1, \ldots, A'_s)$ we have $\mathrm{id} \in A'_i$ for $1 \leq i \leq s-1$ and $g \in A'_s$.

If there exists exactly one factorization of g with respect to α, the sandwich is unique.

Proof. In order to compute this sandwich, first compute a factorization of g with respect to α. Let $(y_1, \ldots, y_s)$ be the indices of this factorization, i.e. $g = \alpha[1][y_1] \cdots \alpha[s][y_s]$.

Now let $\beta := \alpha^t$ a sandwich of α with

$$\begin{aligned} t_1 &= \mathrm{id}, \\ t_2 &= \alpha[1][y_1]^{-1}, \\ t_3 &= \alpha[2][y_2]^{-1} \cdot \alpha[1][y_1]^{-1}, \\ &\vdots \\ t_s &= \alpha[s-1][y_{s-1}]^{-1} \cdots \alpha[1][y_1]^{-1}, \\ t_{s+1} &= \mathrm{id}\,. \end{aligned}$$

The resulting block sequence has the properties mentioned above. By multiplying $t_2 = \alpha[1][y_1]^{-1}$ onto the elements in A_1 from the right, this generates id in the first block. By multiplying $t_2^{-1} = \alpha[1][y_1]$ onto the elements in A_2 from the left, the second block contains the element $\alpha[1][y_1] \cdot \alpha[2][y_2]$. Between the second and the third block, the element $t_3 = \alpha[2][y_2]^{-1} \cdot \alpha[1][y_1]^{-1}$ is multiplied onto the elements in the second block from the right, which precisely cancels out the element $\alpha[1][y_1] \cdot \alpha[2][y_2]$, and so on. Thus the first $s-1$ blocks of β contain id.

In the very last step, the element $t_s^{-1} = \alpha[1][y_1] \cdots \alpha[s-1][y_{s-1}]$ is multiplied onto the elements in block A_s from the left. Thus the last block contains the element $\alpha[1][y_1] \cdots \alpha[s-1][y_{s-1}] \cdot \alpha[s][y_s] = g$, as asserted.

If there exists exactly one factorization of g with respect to α, the $y_1, \ldots, y_s$ are determined uniquely, and thus the $t_1, \ldots, t_{s+1}$ are determined uniquely. □

In contrast to the normalization in Section 5.1.5 (which creates an ℓ-canonical block sequence), id now does not necessarily appear in the first positions in the blocks anymore.

Also, a g-normalization requires being able to compute a factorization of g with respect to α. This is not required when performing a usual normalization (Section 5.1.5).

Example 5.10. Let $G = D_{2\cdot 9}$, $\alpha = \begin{pmatrix} \sigma^5 \\ \sigma^5\tau \\ \sigma^7\tau \end{pmatrix} \begin{pmatrix} \sigma^8 \\ \sigma^5 \\ \sigma^2 \end{pmatrix} \begin{pmatrix} \sigma^4\tau \\ \sigma^2 \end{pmatrix} \in \Lambda(G)$, and $g = \sigma^8$. The factorization of g with respect to α is

$$\begin{pmatrix} \sigma^5 \\ \boxed{\sigma^5\tau} \\ \sigma^7\tau \end{pmatrix} \begin{pmatrix} \sigma^8 \\ \sigma^5 \\ \boxed{\sigma^2} \end{pmatrix} \begin{pmatrix} \boxed{\sigma^4\tau} \\ \sigma^2 \end{pmatrix}.$$

With this, we can perform the σ^8-normalization of α:

$$\begin{pmatrix} \sigma^5 \\ \sigma^5\tau \\ \sigma^7\tau \end{pmatrix} \sigma^5\tau \cdot \sigma^5\tau \begin{pmatrix} \sigma^8 \\ \sigma^5 \\ \sigma^2 \end{pmatrix} \sigma^{-2}\sigma^5\tau \cdot \sigma^5\tau\sigma^2 \begin{pmatrix} \sigma^4\tau \\ \sigma^2 \end{pmatrix}$$
$$\doteq \begin{pmatrix} \sigma\tau \\ \mathrm{id} \\ \sigma^2 \end{pmatrix} \begin{pmatrix} \sigma^3 \\ \sigma^6 \\ \mathrm{id} \end{pmatrix} \begin{pmatrix} \sigma^8 \\ \sigma\tau \end{pmatrix} =: \beta.$$

Indeed, the first blocks contain id and the last block contains $g = \sigma^8$.

As sandwiches never change factorization indices, we can obtain the factorization of σ^8 in β by selecting the elements at the same indices as in α. By construction, this selects id in the first blocks and g in the last block. In the β above, we can very easily verify that $\mathrm{id} \cdot \mathrm{id} \cdot \sigma^8 = \sigma^8$.

Corollary 5.11. *For every $\alpha = (A_1, \ldots, A_s) \in \Xi_s(G)$ with $\mathrm{id} \in A_1 \cdots A_s$ there exists a sandwich β of α, such that every block of β contains* id.

Proof. An id-normalization computes the sandwich. □

Corollary 5.12. *For every $\alpha = (A_1, \ldots, A_s) \in \Xi_{(r_1,\ldots,r_s)}(G)$ with $\mathrm{id} \in A_1 \cdots A_s$ there exist $\mathfrak{s}_{(x_1,\ldots,x_{s-1})} \in \mathfrak{S}_s(G)$ and $\mathfrak{e}_{(\pi_1,\ldots,\pi_s)} \in \mathfrak{E}_{(r_1,\ldots,r_s)}(G)$ such that $\mathfrak{e}_{(\pi_1,\ldots,\pi_s)}(\mathfrak{s}_{(x_1,\ldots,x_{s-1})}(\alpha))$ is a canonical block sequence.*

Proof. Choose the permutations $\pi_1, \ldots, \pi_s$ in such a way that they move id to the top within every block of the id-normalization of α. □

Note that in Corollary 5.11 we have $\breve{\beta} = \breve{\alpha}$ by Proposition 5.6. In contrast, in Corollary 5.12 we do not necessarily have $\breve{\beta} = \breve{\alpha}$, due to the element shuffle; see Section 5.1.1 for details.

Generalization: (i, g)-normalizations. The transformation above can be generalized to an (i, g)-normalization ($1 \leq i \leq s$ and $g \in A_1 \cdots A_s$), where in the resulting block sequence the ith block contains the element g and all other blocks contain id.

In order to compute this, perform a g-normalization only up to block i and then similarly from the right end back to the ith block.

Precisely, let $(y_1, \ldots, y_s)$ be the indices of a factorization of a $g \in A_1 \cdots A_s$ with respect to a block sequence $\alpha = (A_1, \ldots, A_s) \in \Xi_s(G)$, then an (i, g)-normalization applies the following sandwich (which is uniquely determined by i and g, if there exists exactly one factorization of g with respect to α) to α:

$$\begin{aligned}
t_1 &= \mathrm{id},\\
t_2 &= \alpha[1][y_1]^{-1},\\
t_3 &= \alpha[2][y_2]^{-1} \cdot \alpha[1][y_1]^{-1},\\
&\vdots\\
t_i &= \alpha[i-1][y_{i-1}]^{-1} \cdots \alpha[1][y_1]^{-1},\\
t_{i+1} &= \alpha[i+1][y_{i+1}] \cdots \alpha[s][y_s],\\
&\vdots\\
t_{s-1} &= \alpha[s-1][y_{s-1}] \cdot \alpha[s][y_s],\\
t_s &= \alpha[s][y_s],\\
t_{s+1} &= \mathrm{id}\,.
\end{aligned}$$

Analogously to above, it can be verified that the resulting block sequence indeed has the asserted properties.

Note that just like a g-normalization, an (i, g)-normalization requires the knowledge of a factorization of g with respect to α.

A g-normalization is nothing else than an (s, g)-normalization.

Example 5.13. Let $G = D_{2\cdot 4}$, $\alpha = \binom{\sigma^2\tau}{\sigma^3}\binom{\sigma\tau}{\tau}\binom{\sigma^3\tau}{\sigma} \in \Lambda(G)$, and $g = \sigma^3\tau$. The factorization of g is $g = \sigma^3 \cdot \sigma\tau \cdot \sigma$. A $(2, \sigma^3\tau)$-normalization results in

$$\begin{aligned}
&\binom{\sigma^2\tau}{\sigma^3}\sigma^{-3} \cdot \sigma^3\binom{\sigma\tau}{\tau}\sigma \cdot \sigma^{-1}\binom{\sigma^3\tau}{\sigma}\\
\breve{=}\ &\binom{\sigma\tau}{\mathrm{id}}\binom{\sigma^3\tau}{\sigma^2\tau}\binom{\sigma^2\tau}{\mathrm{id}}.
\end{aligned}$$

5.1.7. Fusing / Refinement

Two adjacent blocks A_i and A_{i+1} of a logarithmic signature $\alpha = (A_1, A_2, \ldots, A_s)$ can be *fused* to a single block containing the elements $\{g \cdot h \mid g \in A_i, h \in A_{i+1}\}$. As the two small

blocks A_i and A_{i+1} locally generate the same elements as the large fused block, the result of course is a logarithmic signature α'. α is called a *refinement* of α'.

5.1.8. Block Substitution

Adjacent blocks $A_i, A_{i+1}, \ldots, A_{i+k}$ of a logarithmic signature $\alpha = (A_1, A_2, \ldots, A_s)$ can be replaced by new blocks $A'_j, A'_{j+1}, \ldots, A'_{j+m}$, provided that they (locally) generate the same elements, i.e.

$$\begin{aligned} &\{a_i \cdot a_{i+1} \cdots a_{i+k} \mid a_i \in A_i, a_{i+1} \in A_{i+1}, \ldots, a_{i+k} \in A_{i+k}\} \\ = &\{a_j \cdot a_{j+1} \cdots a_{j+m} \mid a_j \in A'_j, a_{j+1} \in A'_{j+1}, \ldots, a_{j+m} \in A'_{j+m}\} \end{aligned}$$

and

$$|A_i| \cdot |A_{i+1}| \cdots |A_{i+k}| = |A'_j| \cdot |A'_{j+1}| \cdots |A'_{j+m}|.$$

The first property ensures that the new blocks generate the same elements as the old ones, and the second property additionally ensures that logarithmic signatures are mapped to logarithmic signatures (each element has a *unique* factorization).

This is a very generic transformation, which highly depends on the group G. Element shuffles, block shuffles, fusing and refinement are block substitutions.

Efficiency. Although other operations can be expressed using series of block substitutions, in practice (during an algorithm involving random block substitutions, like LS-Gen in Section 6.5) for efficiency it can be useful to perform the other operations anyway and not rely on block substitutions only.

For example, let $G = \mathbb{Z}_{2^n}$ and $\alpha = (A_1, A_2, \ldots, A_n) \in \Lambda(G)$ canonical with $|A_i| = 2$ for all i. Assume that we apply r block substitutions that only exchange two adjacent blocks (at random positions). By Lemma 8.4, there is exactly one $g \in E(\alpha)$ with $\mathrm{ord}(g) = 2^n$ and we can assume it is in A_1. What is the expected value of r such that the element of order 2^n reaches the rightmost block? Obviously, $n - 1$ block exchanges theoretically are sufficient, but in practice the block will usually move back and forth a few times before finally reaching the right side.

If every block substitution would affect the current block containing the element of order 2^n, this can be interpreted as an undirected random walk, thus the expected number of required block substitutions is about n^2 [Mot95]. However, only about 2 out of n block substitutions actually affect the block with the element of order 2^n. Thus, all in all about $\frac{n^3}{2}$ block substitutions are expected to be required for the element of order 2^n reaching the rightmost block.

Block shuffles achieve the same effect a lot more efficiently (a block shuffle that exchanges A_1 with A_n achieves it in one step).

Implementation. Random block substitutions can be implemented using the two functions below. This approach works for all groups (including non-abelian) and no

structural knowledge on the group is required. However, when knowing more about the structure of the group and the possible effects of block substitutions, implementing these effects directly will be more efficient.

Function `Subst`(BlockSequence α) : BlockSequence

1. Let F be a list of all elements generated by α (without duplicates).
   If id $\notin F$ or $|F| \neq |\alpha[1]| \cdots |\alpha[|\alpha|]|$: abort.
2. Remove id from F and permute the elements in F randomly.
3. Let $P := [p_1, \ldots, p_r]$ be the multiset of all prime factors in the numbers $|\alpha[i]|$ for all i (including duplicate prime factors).
4. Repeat:
   a) Choose a random $s \in \{1, \ldots, r\}$ and partition P randomly into multisets $P_1 = [p_{i_{1,1}}, \ldots, p_{i_{1,n_1}}], \ldots, P_s = [p_{i_{s,1}}, \ldots, p_{i_{s,n_s}}]$.
   Let α' be a new block sequence of type $t(\alpha') = (\prod_{j=1}^{n_1} p_{i_{1,j}}, \ldots, \prod_{j=1}^{n_s} p_{i_{s,j}})$, containing only id elements.
   b) If `SubstFillRec`$(\alpha', 1, 1, (\mathrm{id}), F)$: `return` α'.

Function `SubstFillRec`(BlockSequence α, Int x, Int y, List<GroupElement> L, List<GroupElement> F) : Bool

1. If $x = |\alpha| + 1$: `return` *true*.
2. If $y = 1$: `return SubstFillRec`$(\alpha, x, y + 1, L, F)$.
3. For each $c \in F$ (from left to right):
   a) If $c < \alpha[x][y - 1]$: `continue`.
   b) Set $\alpha[x][y] \leftarrow c$, $F' \leftarrow F$ and $v \leftarrow$ *true*.
   c) For each $l \in L$:
      i. Let $n \leftarrow l \cdot c$.
      ii. If $n \notin F'$: set $v \leftarrow$ *false* and `break`.
      iii. Set $F' \leftarrow F' \setminus \{n\}$.
   d) If not v: `continue`.
   e) If $y = |\alpha[x]|$:
      i. Let $L' \leftarrow L \cdot \alpha[x]$ (product of all elements in L and block $\alpha[x]$).
      ii. If `SubstFillRec`$(\alpha, x + 1, 1, L', F')$: `return` *true*.
   else if `SubstFillRec`$(\alpha, x, y + 1, L, F')$: `return` *true*.
4. `Return` *false*.

Comments.

- The input for `Subst` is expected to be a block sequence that is generating id and is not generating duplicates. The output is a canonical block sequence generating exactly the same elements as the input block sequence. Additionally, the elements in the blocks of the output block sequence are in ascending order (the algorithm requires such an ordering of group elements to exist, and id must be the lowest element).

  The algorithm recursively moves from the left to the right in the blocks of α (index variable x) and from top to bottom in each block (index variable y).

  The list F always contains possible candidates for the current position. F is updated (forming a new F') every time when a candidate element is assigned.

  L is a list of elements generated by all blocks left of the current one (i.e. all possible products of elements in the blocks $1, \ldots, x-1$).

- At the start of the loop in `Subst`, new block sizes are computed. This is done by collecting all prime factors from the sizes of the blocks of α and composing them randomly in a new way.

  For example, if the input blocks have the sizes $(4, 6, 10)$, we would get $P = [2, 2, 2, 3, 2, 5]$. s could be chosen randomly as 4 and the random partitions of P as $P_1 = [5, 2], P_2 = [3], P_3 = [2], P_4 = [2, 2]$. With these, we would get

  $$\alpha' = ((\mathrm{id}, \mathrm{id}, \mathrm{id}, \mathrm{id}, \mathrm{id}, \mathrm{id}, \mathrm{id}, \mathrm{id}, \mathrm{id}, \mathrm{id}), (\mathrm{id}, \mathrm{id}, \mathrm{id}), (\mathrm{id}, \mathrm{id}), (\mathrm{id}, \mathrm{id}, \mathrm{id}, \mathrm{id})).$$

- The loop in `Subst` choosing new block sizes is required, because there may be types that do not permit valid factorizations.

  For example, let $G = D_{2 \cdot 6}$, $A_1 := (\mathrm{id}, \sigma, \sigma^4\tau)$, $A_2 := (\mathrm{id}, \tau)$ and $A := A_1 A_2$, then there are no blocks A_1' and A_2' with $A_1' A_2' = A$, $|A_1'| = 2$ and $|A_2'| = 3$, i.e. only block sequences of type $(3, 2)$ can be factorizations of A, not ones of type $(2, 3)$.

  For dihedral groups, we have a look at block substitutions in the Sections 9.1.1 and 9.1.2.

- The check for $c < \alpha[x][y-1]$ is a performance optimization. It ensures that the elements in the current block are ordered ascendingly. For example, the blocks $(0, 1, 3)$ and $(0, 3, 1)$ contain exactly the same elements and it does not make any sense to try $(0, 3, 1)$ when the block $(0, 1, 3)$ cannot lead to a valid factorization (because then $(0, 3, 1)$ also will not lead to a valid factorization).

- Only elements in F, which initially contains all elements generated by α except id, are considered to be candidates for the current position. This only restricts generated block sequences up to element shuffle-sandwiches (i.e. applying an element shuffle and a sandwich transformation). *All* possible block substitutions can be obtained by computing random element shuffle-sandwiches of the block sequences generated by the algorithm.

In order to see why this is correct, let $\beta = (B_1, \ldots, B_s) \in \Xi_s(G)$ arbitrary (not necessarily canonical) with $\prod_{i=1}^{s} |B_i| = |F| + 1$ and $B_1 \cdots B_s = F \cup \{\mathrm{id}\}$. By applying an id-normalization to β (Section 5.1.6; note that $\mathrm{id} \in B_1 \cdots B_s$, thus this normalization can be applied), we obtain a block sequence β' where every block contains id, and thus $E(\beta') \subseteq F \cup \{\mathrm{id}\}$. By sorting the elements within each block, we obtain a canonical block sequence.

The algorithm generates all these canonical, sorted block sequences.

- The algorithm theoretically could be used to generate whole logarithmic signatures for groups. However, it is too slow when the number of blocks/elements is large.

5.1.9. Selective Shift

Let G be a group, $\alpha = (A_1, \ldots, A_n) \in \Lambda(G)$, $j, k \in \mathbb{N}$ with $1 \leq j \leq k \leq n$. Let $A := A_j \cdots A_k$.

If A is r-periodic, let $A = R \cdot S$ with $R \subseteq G$ and $S \leq G$ (where S contains id and the periods of A, see Lemma 2.7). Multiplying any $s \in S$ to an element in A_{k+1} from the left clearly results in another logarithmic signature for G and is called an *r-selective shift*.

If A is ℓ-periodic, let $A = S \cdot R$ with $R \subseteq G$ and $S \leq G$. Multiplying any $s \in S$ to an element in A_{j-1} from the right results in another logarithmic signature for G and is called an *ℓ-selective shift*.

If A is periodic with either $A = R \cdot S$ or $A = S \cdot R$ (with R and S like above) and S is a normal subgroup, every element $s \in S$ can freely be multiplied to any elements in the blocks $A_1, \ldots, A_{j-1}, A_{k+1}, \ldots, A_n$ (if S is normal, it can be swapped with all other blocks and r- and ℓ-selective shifts can be performed in each position). We call this a *selective shift*.

If G is abelian, all subgroups are normal, i.e. elements of S can always be multiplied onto elements in other blocks.

5.1.10. Automorphism

Let $\varphi : G \to G$ be an automorphism. Then $\varphi(\alpha)$ is another logarithmic signature of G. This is proven in a more general form (for surjective homomorphisms and multiple factorizations) in Theorem 7.3.

Factoring. The factorization index vector of an element $g \in G$ with respect to α is exactly the same as the one for $\varphi(g)$ with respect to $\varphi(\alpha)$.

Example 5.14. Inspired by elementary row additions in matrices, a component addition automorphism can be defined for some groups.

Let $G = L \times S \times H$ with L abelian (i.e. $L \times \{\mathrm{id}\} \times \{\mathrm{id}\} \subseteq Z(G)$) and $\psi : S \to L$ a homomorphism. Then

$$\varphi : G \to G : (l, s, h) \mapsto (l \cdot \psi(s), s, h)$$

is an automorphism.

A generalization of this are central automorphisms.

Proof. Clearly, φ is bijective. We have $\varphi^{-1} : G \to G : (l, s, h) \mapsto (l \cdot \psi(s^{-1}), s, h)$ (due to $(l\psi(s))\psi(s^{-1}) = l\psi(s)\psi(s^{-1}) = l\psi(ss^{-1}) = l$).

φ is a homomorphism. Let $g_1 = (l_1, s_1, h_1), g_2 = (l_2, s_2, h_2) \in G$.

$$\begin{aligned}\varphi(g_1 \cdot g_2) &= \varphi((l_1, s_1, h_1) \cdot (l_2, s_2, h_2)) = \varphi((l_1 l_2, s_1 s_2, h_1 h_2)) = (l_1 l_2 \psi(s_1 s_2), s_1 s_2, h_1 h_2) \\ &= (l_1 l_2 \psi(s_1)\psi(s_2), s_1 s_2, h_1 h_2) = (l_1 \psi(s_1) l_2 \psi(s_2), s_1 s_2, h_1 h_2) \\ &= (l_1 \psi(s_1), s_1, h_1) \cdot (l_2 \psi(s_2), s_2, h_2) = \varphi((l_1, s_1, h_1)) \cdot \varphi((l_2, s_2, h_2)) \\ &= \varphi(g_1) \cdot \varphi(g_2).\end{aligned}$$

□

Now, let $G = \mathbb{Z}_8 \oplus \mathbb{Z}_2$, $\alpha = \binom{(0,0)}{(1,0)}\binom{(0,0)}{(2,0)}\binom{(0,0)}{(4,0)}\binom{(0,0)}{(4,1)} \in \Lambda(G)$, $\psi : \mathbb{Z}_2 \to \mathbb{Z}_8 : y \mapsto 4y$, $\varphi : G \to G : (x, y) \mapsto (x + 4y, y)$, and $g = (2, 1)$. The factorization index vectors for g with respect to α and $\varphi(g)$ with respect to $\varphi(\alpha)$ are the same:

$$\alpha = \begin{pmatrix}\boxed{(0,0)}\\(1,0)\end{pmatrix}\begin{pmatrix}(0,0)\\\boxed{(2,0)}\end{pmatrix}\begin{pmatrix}(0,0)\\\boxed{(4,0)}\end{pmatrix}\begin{pmatrix}(0,0)\\\boxed{(4,1)}\end{pmatrix} \text{ (sum of selected elements} = (2,1) = g),$$

$$\varphi(\alpha) = \begin{pmatrix}\boxed{(0,0)}\\(1,0)\end{pmatrix}\begin{pmatrix}(0,0)\\\boxed{(2,0)}\end{pmatrix}\begin{pmatrix}(0,0)\\\boxed{(4,0)}\end{pmatrix}\begin{pmatrix}(0,0)\\\boxed{(0,1)}\end{pmatrix} \text{ (sum of selected elements} = (6,1) = \varphi(g)).$$

Let $\mathfrak{a}_\varphi : \Xi(G) \to \Xi(G) : \alpha \mapsto \varphi(\alpha)$ be the transformation that applies the automorphism $\varphi \in \mathrm{Aut}(G)$ to all elements in α. Let $\mathfrak{A}(G) := \{\mathfrak{a}_\varphi \mid \varphi \in \mathrm{Aut}(G)\}$ be the set of all automorphism application transformations.

Together with composition as binary operation, $\mathfrak{A}(G)$ is a group. We have $\mathfrak{a}_{\widetilde{\varphi}} \circ \mathfrak{a}_\varphi = \mathfrak{a}_{\widetilde{\varphi} \circ \varphi}$, the identity is $\mathfrak{a}_{\mathrm{id}}$, and $\mathfrak{a}_\varphi^{-1} = \mathfrak{a}_{\varphi^{-1}}$.

$\mathfrak{A}(G)$ is abelian if and only if $\mathrm{Aut}(G)$ is abelian.

5.1.11. Combining Sandwich and Normalization

When starting with a canonical logarithmic signature (of type $(r_1, \ldots, r_s)$) and first applying a sandwich transformation $\mathfrak{s}_{(x_1,\ldots,x_{s-1})}$ and subsequently normalizing the signature, we get exactly the original canonical logarithmic signature again, i.e. the normalization undoes the sandwich:

$$\begin{pmatrix}\mathrm{id}\\a_{1,2}\\\vdots\\a_{1,r_1}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\a_{2,2}\\\vdots\\a_{2,r_2}\end{pmatrix}\cdots\begin{pmatrix}\mathrm{id}\\a_{s,2}\\\vdots\\a_{s,r_s}\end{pmatrix}$$

$$\xrightarrow{\mathfrak{s}_{(x_1,\ldots,x_{s-1})}} \begin{pmatrix}\mathrm{id}\cdot x_1\\a_{1,2}\cdot x_1\\\vdots\\a_{1,r_1}\cdot x_1\end{pmatrix}\begin{pmatrix}x_1^{-1}\cdot \mathrm{id}\cdot x_2\\x_1^{-1}\cdot a_{2,2}\cdot x_2\\\vdots\\x_1^{-1}\cdot a_{2,r_2}\cdot x_2\end{pmatrix}\cdots\begin{pmatrix}x_{s-1}^{-1}\cdot \mathrm{id}\\x_{s-1}^{-1}\cdot a_{s,2}\\\vdots\\x_{s-1}^{-1}\cdot a_{s,r_s}\end{pmatrix}$$

$$\xrightarrow{\text{Normalization}} \begin{pmatrix} \mathrm{id} \cdot x_1 \cdot x_1^{-1} \\ a_{1,2} \cdot x_1 \cdot x_1^{-1} \\ \vdots \\ a_{1,r_1} \cdot x_1 \cdot x_1^{-1} \end{pmatrix} \begin{pmatrix} x_1 \cdot x_1^{-1} \cdot \mathrm{id} \cdot x_2 \cdot x_2^{-1} \\ x_1 \cdot x_1^{-1} \cdot a_{2,2} \cdot x_2 \cdot x_2^{-1} \\ \vdots \\ x_1 \cdot x_1^{-1} \cdot a_{2,r_2} \cdot x_2 \cdot x_2^{-1} \end{pmatrix} \cdots \begin{pmatrix} x_{s-1} \cdot x_{s-1}^{-1} \cdot \mathrm{id} \\ x_{s-1} \cdot x_{s-1}^{-1} \cdot a_{s,2} \\ \vdots \\ x_{s-1} \cdot x_{s-1}^{-1} \cdot a_{s,r_s} \end{pmatrix}$$
$$= \begin{pmatrix} \mathrm{id} \\ a_{1,2} \\ \vdots \\ a_{1,r_1} \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ a_{2,2} \\ \vdots \\ a_{2,r_2} \end{pmatrix} \cdots \begin{pmatrix} \mathrm{id} \\ a_{s,2} \\ \vdots \\ a_{s,r_s} \end{pmatrix}.$$

5.1.12. Combining Translation and Normalization

Left-translating a canonical logarithmic signature by $t \in G$ and subsequently normalizing it effectively results in conjugating all elements by t and a right translation by t:

$$\begin{pmatrix} \mathrm{id} \\ a_{1,2} \\ \vdots \\ a_{1,r_1} \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ a_{2,2} \\ \vdots \\ a_{2,r_2} \end{pmatrix} \cdots \begin{pmatrix} \mathrm{id} \\ a_{s,2} \\ \vdots \\ a_{s,r_s} \end{pmatrix}$$
$$\xrightarrow{\text{Left translation}} \begin{pmatrix} t \cdot \mathrm{id} \\ t \cdot a_{1,2} \\ \vdots \\ t \cdot a_{1,r_1} \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ a_{2,2} \\ \vdots \\ a_{2,r_2} \end{pmatrix} \cdots \begin{pmatrix} \mathrm{id} \\ a_{s,2} \\ \vdots \\ a_{s,r_s} \end{pmatrix}$$
$$\xrightarrow{\text{Normalization}} \begin{pmatrix} t \cdot \mathrm{id} \cdot t^{-1} \\ t \cdot a_{1,2} \cdot t^{-1} \\ \vdots \\ t \cdot a_{1,r_1} \cdot t^{-1} \end{pmatrix} \begin{pmatrix} t \cdot \mathrm{id} \cdot t^{-1} \\ t \cdot a_{2,2} \cdot t^{-1} \\ \vdots \\ t \cdot a_{2,r_2} \cdot t^{-1} \end{pmatrix} \cdots \begin{pmatrix} t \cdot \mathrm{id} \\ t \cdot a_{s,2} \\ \vdots \\ t \cdot a_{s,r_s} \end{pmatrix}$$
$$= \begin{pmatrix} \mathrm{id} \\ t \cdot a_{1,2} \cdot t^{-1} \\ \vdots \\ t \cdot a_{1,r_1} \cdot t^{-1} \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ t \cdot a_{2,2} \cdot t^{-1} \\ \vdots \\ t \cdot a_{2,r_2} \cdot t^{-1} \end{pmatrix} \cdots \begin{pmatrix} t \cdot \mathrm{id} \\ t \cdot a_{s,2} \\ \vdots \\ t \cdot a_{s,r_s} \end{pmatrix}$$
$$\xrightarrow{\text{Extract right translation}} \begin{pmatrix} \mathrm{id} \\ t \cdot a_{1,2} \cdot t^{-1} \\ \vdots \\ t \cdot a_{1,r_1} \cdot t^{-1} \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ t \cdot a_{2,2} \cdot t^{-1} \\ \vdots \\ t \cdot a_{2,r_2} \cdot t^{-1} \end{pmatrix} \cdots \begin{pmatrix} t \cdot \mathrm{id} \cdot t^{-1} \\ t \cdot a_{s,2} \cdot t^{-1} \\ \vdots \\ t \cdot a_{s,r_s} \cdot t^{-1} \end{pmatrix} (t)$$
$$= \begin{pmatrix} \mathrm{id} \\ t \cdot a_{1,2} \cdot t^{-1} \\ \vdots \\ t \cdot a_{1,r_1} \cdot t^{-1} \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ t \cdot a_{2,2} \cdot t^{-1} \\ \vdots \\ t \cdot a_{2,r_2} \cdot t^{-1} \end{pmatrix} \cdots \begin{pmatrix} \mathrm{id} \\ t \cdot a_{s,2} \cdot t^{-1} \\ \vdots \\ t \cdot a_{s,r_s} \cdot t^{-1} \end{pmatrix} (t).$$

5.1.13. Combining Translation, Element Shuffle and Normalization (TSN)

We call performing the following transformations onto a logarithmic signature α for a group G a *TSN transformation*:

1. Left-translate α by an element $t \in G$.
2. Shuffle the elements within all blocks of α randomly.
3. Normalize α.
4. Extract the right translation (from the last block of α) and discard it.

Example 5.15. Let $G = D_{2\cdot 8}$ and $t = \sigma^6$.

$$
\begin{aligned}
& \binom{\mathrm{id}}{\sigma^1}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\tau} \\
\xrightarrow{\text{Left translation}}\ & \sigma^6\binom{\mathrm{id}}{\sigma^1}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\tau} \\
\overset{\smile}{=}\ & \binom{\sigma^6}{\sigma^7}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\tau} \\
\xrightarrow{\text{Random element shuffle}}\ & \binom{\sigma^7}{\sigma^6}\binom{\mathrm{id}}{\sigma^2}\binom{\sigma^4}{\mathrm{id}}\binom{\tau}{\mathrm{id}} \\
\xrightarrow{\text{Normalization}}\ & \binom{\sigma^7}{\sigma^6}\sigma^1\cdot\sigma^7\binom{\mathrm{id}}{\sigma^2}\sigma^1\cdot\sigma^7\binom{\sigma^4}{\mathrm{id}}\sigma^5\cdot\sigma^3\binom{\tau}{\mathrm{id}} \\
\overset{\smile}{=}\ & \binom{\mathrm{id}}{\sigma^7}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma^4}\binom{\sigma^3\tau}{\sigma^3} \\
\xrightarrow{\text{Extract right translation}}\ & \binom{\mathrm{id}}{\sigma^7}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\sigma^6\tau}\cdot\sigma^3\tau \\
\xrightarrow{\text{Discard right translation}}\ & \binom{\mathrm{id}}{\sigma^7}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\sigma^6\tau}.
\end{aligned}
$$

Factoring. When being able to factor elements in the original logarithmic signature, one can derive factorizations for elements in the transformed logarithmic signature. Let $t \in G$ be the left translation and $r \in G$ the right translation being discarded at the end. In order to get the factorization for an element $x \in G$ with respect to the TSN-transformed logarithmic signature, factor $x' := t^{-1}xr$ in the original logarithmic signature and apply the transformation. The translations and the normalization do not change element selection indices; the element shuffle does.

Example 5.16. We want to factor $x = \sigma^7\tau$ in the TSN-transformed logarithmic signature from the previous example. Here, $t = \sigma^6$ and $r = \sigma^3\tau$. Thus, $x' = t^{-1}xr = \sigma^2 \cdot \sigma^7\tau \cdot \sigma^3\tau = \sigma^6$. In the original, exact transversal logarithmic signature, factorizations are easy to compute.

$$
\binom{\boxed{\mathrm{id}}}{\sigma^1}\binom{\mathrm{id}}{\boxed{\sigma^2}}\binom{\mathrm{id}}{\boxed{\sigma^4}}\binom{\boxed{\mathrm{id}}}{\tau} \qquad \text{(fac. of } t^{-1}xr\text{)}
$$

$$\xrightarrow{\text{Left translation}} \left(\boxed{\sigma^6}\ \sigma^7\right)\left(\mathrm{id}\ \boxed{\sigma^2}\right)\left(\mathrm{id}\ \boxed{\sigma^4}\right)\left(\boxed{\mathrm{id}}\ \tau\right) \qquad \text{(fac. of } xr\text{)}$$

$$\xrightarrow{\text{Random element shuffle}} \left(\sigma^7\ \boxed{\sigma^6}\right)\left(\mathrm{id}\ \boxed{\sigma^2}\right)\left(\boxed{\sigma^4}\ \mathrm{id}\right)\left(\tau\ \boxed{\mathrm{id}}\right) \qquad \text{(fac. of } xr\text{)}$$

$$\xrightarrow{\text{Normalization}} \left(\mathrm{id}\ \boxed{\sigma^7}\right)\left(\mathrm{id}\ \boxed{\sigma^2}\right)\left(\boxed{\mathrm{id}}\ \sigma^4\right)\left(\sigma^3\tau\ \boxed{\sigma^3}\right) \qquad \text{(fac. of } xr\text{)}$$

$$\xrightarrow{\text{Extract right translation}} \left(\mathrm{id}\ \boxed{\sigma^7}\right)\left(\mathrm{id}\ \boxed{\sigma^2}\right)\left(\boxed{\mathrm{id}}\ \sigma^4\right)\left(\mathrm{id}\ \boxed{\sigma^6\tau}\right)\cdot\boxed{\sigma^3\tau} \qquad \text{(fac. of } xr\text{)}$$

$$\xrightarrow{\text{Discard right translation}} \left(\mathrm{id}\ \boxed{\sigma^7}\right)\left(\mathrm{id}\ \boxed{\sigma^2}\right)\left(\boxed{\mathrm{id}}\ \sigma^4\right)\left(\mathrm{id}\ \boxed{\sigma^6\tau}\right) \qquad \text{(fac. of } x\text{).}$$

This indeed is the factorization of $x = \sigma^7\tau$, which we were looking for.

Implementation. The TSN transformation can be implemented efficiently:

1. Multiply all the elements in the first block by t from the left.
2. Shuffle the elements within all blocks.
3. For $i \leftarrow 1$ to $|\alpha| - 1$:
   - Let $z \leftarrow \alpha[i][1]$. Multiply all elements in $\alpha[i]$ by z^{-1} from the right and all elements in $\alpha[i+1]$ by z from the left.
4. Multiply all elements in $\alpha[|\alpha|]$ by $\alpha[|\alpha|][1]^{-1}$ from the right.

Sandwich ineffectiveness. It is tempting to insert a sandwich transformation after the left translation to further increase flexibility and complexity. However, a sandwich transformation here would be ineffective, because it is removed by the normalization:

1. We start with a block sequence $\alpha = (A_1, \ldots, A_s)$ of type $t(\alpha) = (r_1, \ldots, r_s)$ with $A_i = (a_{i,1}, \ldots, a_{i,r_i})$ for $1 \le i \le s$, and apply a left translation using $t_1^{-1} \in G$ and a sandwich using $(t_2, \ldots, t_s) \in G^{s-1}$. We obtain new blocks $A_1^{(1)}, \ldots, A_s^{(1)}$:

$$\begin{aligned} A_1^{(1)} &= (t_1^{-1}a_{1,1}t_2, \ldots, t_1^{-1}a_{1,r_1}t_2),\\ &\vdots\\ A_{s-1}^{(1)} &= (t_{s-1}^{-1}a_{s-1,1}t_s, \ldots, t_{s-1}^{-1}a_{s-1,r_{s-1}}t_s),\\ A_s^{(1)} &= (t_s^{-1}a_{s,1}, \ldots, t_s^{-1}a_{s,r_s}). \end{aligned}$$

2. We apply an element shuffle using the permutations $\pi_1 \in \mathrm{Sym}(r_1), \ldots, \pi_s \in \mathrm{Sym}(r_s)$:

$$\begin{aligned} A_1^{(2)} &= (t_1^{-1}a_{1,\pi_1(1)}t_2, \ldots, t_1^{-1}a_{1,\pi_1(r_1)}t_2),\\ &\vdots\\ A_{s-1}^{(2)} &= (t_{s-1}^{-1}a_{s-1,\pi_{s-1}(1)}t_s, \ldots, t_{s-1}^{-1}a_{s-1,\pi_{s-1}(r_{s-1})}t_s),\\ A_s^{(2)} &= (t_s^{-1}a_{s,\pi_s(1)}, \ldots, t_s^{-1}a_{s,\pi_s(r_s)}). \end{aligned}$$

3. Now we perform a normalization. In the first round, the element $t_2^{-1}a_{1,\pi_1(1)}^{-1}t_1$ is multiplied from the right to all elements in the first block, and $t_1^{-1}a_{1,\pi_1(1)}t_2$ is multiplied from the left to all elements in the second block; we get:

$$\begin{aligned}
A_1^{(3)} &= (\mathrm{id}, t_1^{-1}a_{1,\pi_1(2)}a_{1,\pi_1(1)}^{-1}t_1, \ldots, t_1^{-1}a_{1,\pi_1(r_1)}a_{1,\pi_1(1)}^{-1}t_1),\\
A_2^{(3)} &= (t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}t_3, \ldots, t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(r_2)}t_3),\\
A_3^{(3)} &= (t_3^{-1}a_{3,\pi_3(1)}t_4, \ldots, t_3^{-1}a_{3,\pi_3(r_3)}t_4),\\
&\vdots\\
A_{s-1}^{(3)} &= (t_{s-1}^{-1}a_{s-1,\pi_{s-1}(1)}t_s, \ldots, t_{s-1}^{-1}a_{s-1,\pi_{s-1}(r_{s-1})}t_s),\\
A_s^{(3)} &= (t_s^{-1}a_{s,\pi_s(1)}, \ldots, t_s^{-1}a_{s,\pi_s(r_s)}).
\end{aligned}$$

4. In the next round of the normalization process (normalizing the second block), $t_3^{-1}a_{2,\pi_2(1)}^{-1}a_{1,\pi_1(1)}^{-1}t_1$ is multiplied from the right to all elements in the second block, and $t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}t_3$ is multiplied from the left to all elements in the third block; we obtain:

$$\begin{aligned}
A_1^{(4)} &= (\mathrm{id}, t_1^{-1}a_{1,\pi_1(2)}a_{1,\pi_1(1)}^{-1}t_1, \ldots, t_1^{-1}a_{1,\pi_1(r_1)}a_{1,\pi_1(1)}^{-1}t_1),\\
A_2^{(4)} &= (\mathrm{id}, t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(2)}a_{2,\pi_2(1)}^{-1}a_{1,\pi_1(1)}^{-1}t_1, \ldots,\\
&\quad t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(r_2)}a_{2,\pi_2(1)}^{-1}a_{1,\pi_1(1)}^{-1}t_1),\\
A_3^{(4)} &= (t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}a_{3,\pi_3(1)}t_4, \ldots, t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}a_{3,\pi_3(r_3)}t_4),\\
A_4^{(4)} &= (t_4^{-1}a_{4,\pi_4(1)}t_5, \ldots, t_4^{-1}a_{4,\pi_4(r_4)}t_5),\\
&\vdots\\
A_{s-1}^{(4)} &= (t_{s-1}^{-1}a_{s-1,\pi_{s-1}(1)}t_s, \ldots, t_{s-1}^{-1}a_{s-1,\pi_{s-1}(r_{s-1})}t_s),\\
A_s^{(4)} &= (t_s^{-1}a_{s,\pi_s(1)}, \ldots, t_s^{-1}a_{s,\pi_s(r_s)}).
\end{aligned}$$

5. Continuing the normalization process, after the $(s-1)$th round (i.e. after normalizing the $(s-1)$th block), we have:

$$\begin{aligned}
A_1^{(s+1)} &= (\mathrm{id}, t_1^{-1}a_{1,\pi_1(2)}a_{1,\pi_1(1)}^{-1}t_1, \ldots, t_1^{-1}a_{1,\pi_1(r_1)}a_{1,\pi_1(1)}^{-1}t_1),\\
A_2^{(s+1)} &= (\mathrm{id}, t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(2)}a_{2,\pi_2(1)}^{-1}a_{1,\pi_1(1)}^{-1}t_1, \ldots,\\
&\quad t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(r_2)}a_{2,\pi_2(1)}^{-1}a_{1,\pi_1(1)}^{-1}t_1),\\
&\vdots\\
A_{s-1}^{(s+1)} &= (\mathrm{id}, t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}\cdots a_{s-2,\pi_{s-2}(1)}a_{s-1,\pi_{s-1}(2)}a_{s-1,\pi_{s-1}(1)}^{-1}\cdots a_{1,\pi_1(1)}^{-1}t_1,\\
&\quad \ldots,
\end{aligned}$$

$$t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}\cdots a_{s-2,\pi_{s-2}(1)}a_{s-1,\pi_{s-1}(r_{s-1})}a^{-1}_{s-1,\pi_{s-1}(1)}\cdots a^{-1}_{1,\pi_1(1)}t_1),$$
$$A_s^{(s+1)} = (t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}\cdots a_{s,\pi_s(1)},\ldots,t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}\cdots a_{s-1,\pi_{s-1}(1)}a_{s,\pi_s(r_s)}).$$

6. By extracting the right translation $t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}\cdots a_{s,\pi_s(1)}$ and discarding it, we finally obtain:

$$\begin{aligned}
A_1^{(s+2)} &= (\mathrm{id}, t_1^{-1}a_{1,\pi_1(2)}a^{-1}_{1,\pi_1(1)}t_1,\ldots,t_1^{-1}a_{1,\pi_1(r_1)}a^{-1}_{1,\pi_1(1)}t_1),\\
A_2^{(s+2)} &= (\mathrm{id}, t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(2)}a^{-1}_{2,\pi_2(1)}a^{-1}_{1,\pi_1(1)}t_1,\ldots,\\
&\quad t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(r_2)}a^{-1}_{2,\pi_2(1)}a^{-1}_{1,\pi_1(1)}t_1),\\
&\vdots\\
A_{s-1}^{(s+2)} &= (\mathrm{id}, t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}\cdots a_{s-2,\pi_{s-2}(1)}a_{s-1,\pi_{s-1}(2)}a^{-1}_{s-1,\pi_{s-1}(1)}\cdots a^{-1}_{1,\pi_1(1)}t_1,\\
&\quad \ldots,\\
&\quad t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}\cdots a_{s-2,\pi_{s-2}(1)}a_{s-1,\pi_{s-1}(r_{s-1})}a^{-1}_{s-1,\pi_{s-1}(1)}\cdots a^{-1}_{1,\pi_1(1)}t_1),\\
A_s^{(s+2)} &= (\mathrm{id}, t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}\cdots a_{s-1,\pi_{s-1}(1)}a_{s,\pi_s(2)}a^{-1}_{s,\pi_s(1)}\cdots a^{-1}_{1,\pi_1(1)}t_1,\ldots,\\
&\quad t_1^{-1}a_{1,\pi_1(1)}a_{2,\pi_2(1)}\cdots a_{s-1,\pi_{s-1}(1)}a_{s,\pi_s(r_s)}a^{-1}_{s,\pi_s(1)}\cdots a^{-1}_{1,\pi_1(1)}t_1).
\end{aligned}$$

As it can be seen, the $t_2,\ldots,t_s$ from the sandwich transformation were cancelled out during the normalization process, i.e. sandwiching is useless here. Therefore, the definition of the TSN transformation does not contain a sandwich transformation.

Block elements $a_{i,\pi_i(1)}$ are propagated to the right.

Element shuffle. The normalization process always transforms blocks in such a way that the *first* element in a block becomes id. An element shuffle before the normalization has the effect that the block is normalized using an element not necessarily being id, and this element can propagate to the right.

Abelian groups. When the group is abelian, the left translation is ineffective. This can be seen in the explicit formulas above: every element in the final block sequence is translated by $-t_1$ from the left and by t_1 from the right, which cancel out each other.

In every block A_i, all $a_{j,\pi_j(1)}$ cancel out with an $-a_{j,\pi_j(1)}$ for all $1 \le j < i$.

All in all, for abelian groups the TSN transformation is nothing else than a translation of each block by the inverse of an element in this block and an element shuffle; precisely:

$$\begin{aligned}
A'_1 &= (0, a_{1,\pi_1(2)} - a_{1,\pi_1(1)},\ldots,a_{1,\pi_1(r_1)} - a_{1,\pi_1(1)}),\\
A'_2 &= (0, a_{2,\pi_2(2)} - a_{2,\pi_2(1)},\ldots,a_{2,\pi_2(r_2)} - a_{2,\pi_2(1)}),\\
&\vdots\\
A'_s &= (0, a_{s,\pi_s(2)} - a_{s,\pi_s(1)},\ldots,a_{s,\pi_s(r_s)} - a_{s,\pi_s(1)}).
\end{aligned}$$

Corollary 5.17. *Let $G = \mathbb{Z}_2^n$ and $\alpha \in \Lambda(G)$ canonical with $t(\alpha) = (2, \ldots, 2)$. Then α is invariant under TSN transformations.*

Proof. With the above observation, for all $1 \leq i \leq n$ we obtain:

- If $\pi_i = \mathrm{id}$:

$$\begin{aligned} A_i' &= ((0,\ldots,0), a_{i,\pi_i(2)} - a_{i,\pi_i(1)}) = ((0,\ldots,0), a_{i,2} - a_{i,1}) \\ &= ((0,\ldots,0), a_{i,2} - (0,\ldots,0)) = ((0,\ldots,0), a_{i,2}) = A_i. \end{aligned}$$

- If $\pi_i \neq \mathrm{id}$:

$$\begin{aligned} A_i' &= ((0,\ldots,0), a_{i,\pi_i(2)} - a_{i,\pi_i(1)}) = ((0,\ldots,0), a_{i,1} - a_{i,2}) \\ &= ((0,\ldots,0), (0,\ldots,0) - a_{i,2}) = ((0,\ldots,0), -a_{i,2}) \\ &= ((0,\ldots,0), a_{i,2}) = A_i. \end{aligned}$$

□

5.1.14. Combining Translation, Sandwich, Element Shuffle and Automorphism

Recall that $\mathfrak{T}(G)$ is the set of all translation transformations (Section 5.1.3), $\mathfrak{S}_s(G)$ is the set of all sandwich transformations (Section 5.1.4), $\mathfrak{E}_{(r_1,\ldots,r_s)}(G)$ is the set of all element shuffle transformations (Section 5.1.1), and $\mathfrak{A}(G)$ is the set of all automorphism applications (Section 5.1.10). With composition as binary operation, each of these sets is a group.

Proposition 5.18. *The transformations set*

$$\mathfrak{M}_{(r_1,\ldots,r_s)}(G) := \mathfrak{A}(G) \circ \mathfrak{E}_{(r_1,\ldots,r_s)}(G) \circ \mathfrak{S}_s(G) \circ \mathfrak{T}(G)$$

(i.e. all transformations where we first apply a translation, then a sandwich, then an element shuffle and finally an automorphism) is a group.

Especially, every composition of a finite number of translations, sandwiches, element shuffles and automorphism applications can be combined to a composition of one translation, one sandwich, one element shuffle and one automorphism application.

Proof. Let $\alpha = (A_1, \ldots, A_s) = ((a_{1,1}, \ldots, a_{1,r_1}), \ldots, (a_{s,1}, \ldots, a_{s,r_s})) \in \Xi_{(r_1,\ldots,r_s)}(G)$.

Let $u, v, \widetilde{u}, \widetilde{v} \in G$ translation elements, $(z_1, \ldots, z_{s-1}), (\widetilde{z}_1, \ldots, \widetilde{z}_{s-1}) \in G^{s-1}$ sandwich tuples, $(\pi_1, \ldots, \pi_s), (\widetilde{\pi}_1, \ldots, \widetilde{\pi}_s) \in \mathrm{Sym}(r_1) \times \ldots \times \mathrm{Sym}(r_s)$ permutation tuples (for element shuffles), and $\varphi, \widetilde{\varphi}$ automorphisms on G.

$$\begin{aligned} &(\underbrace{(\mathfrak{a}_{\widetilde{\varphi}} \circ \mathfrak{e}_{(\widetilde{\pi}_1,\ldots,\widetilde{\pi}_s)} \circ \mathfrak{s}_{(\widetilde{z}_1,\ldots,\widetilde{z}_{s-1})} \circ \mathfrak{t}_{\widetilde{u},\widetilde{v}})}_{\in \mathfrak{M}_{(r_1,\ldots,r_s)}(G)} \circ \underbrace{(\mathfrak{a}_{\varphi} \circ \mathfrak{e}_{(\pi_1,\ldots,\pi_s)} \circ \mathfrak{s}_{(z_1,\ldots,z_{s-1})} \circ \mathfrak{t}_{u,v})}_{\in \mathfrak{M}_{(r_1,\ldots,r_s)}(G)})(\alpha) \\ = {} &(\mathfrak{a}_{\widetilde{\varphi}} \circ \mathfrak{e}_{(\widetilde{\pi}_1,\ldots,\widetilde{\pi}_s)} \circ \mathfrak{s}_{(\widetilde{z}_1,\ldots,\widetilde{z}_{s-1})} \circ \mathfrak{t}_{\widetilde{u},\widetilde{v}})((\\ &\qquad (\varphi(u a_{1,\pi_1(1)} z_1), \ldots, \varphi(u a_{1,\pi_1(r_1)} z_1)), \end{aligned}$$

$$
\begin{aligned}
&(\varphi(z_1^{-1}a_{2,\pi_2(1)}z_2),\dots,\varphi(z_1^{-1}a_{2,\pi_2(r_2)}z_2)),\\
&\vdots\\
&(\varphi(z_{s-2}^{-1}a_{s-1,\pi_{s-1}(1)}z_{s-1}),\dots,\varphi(z_{s-2}^{-1}a_{s-1,\pi_{s-1}(r_{s-1})}z_{s-1})),\\
&(\varphi(z_{s-1}^{-1}a_{s,\pi_s(1)}v),\dots,\varphi(z_{s-1}^{-1}a_{s,\pi_s(r_s)}v))))\\
=\ \widetilde{\varphi}((&\\
&(\widetilde{u}\varphi(ua_{1,\widetilde{\pi}_1(\pi_1(1))}z_1)\widetilde{z}_1,\dots,\widetilde{u}\varphi(ua_{1,\widetilde{\pi}_1(\pi_1(r_1))}z_1)\widetilde{z}_1),\\
&(\widetilde{z}_1^{-1}\varphi(z_1^{-1}a_{2,\widetilde{\pi}_2(\pi_2(1))}z_2)\widetilde{z}_2,\dots,\widetilde{z}_1^{-1}\varphi(z_1^{-1}a_{2,\widetilde{\pi}_2(\pi_2(r_2))}z_2)\widetilde{z}_2),\\
&\vdots\\
&(\widetilde{z}_{s-2}^{-1}\varphi(z_{s-2}^{-1}a_{s-1,\widetilde{\pi}_{s-1}(\pi_{s-1}(1))}z_{s-1})\widetilde{z}_{s-1},\dots,\\
&\widetilde{z}_{s-2}^{-1}\varphi(z_{s-2}^{-1}a_{s-1,\widetilde{\pi}_{s-1}(\pi_{s-1}(r_{s-1}))}z_{s-1})\widetilde{z}_{s-1}),\\
&(\widetilde{z}_{s-1}^{-1}\varphi(z_{s-1}^{-1}a_{s,\widetilde{\pi}_s(\pi_s(1))}v)\widetilde{v},\dots,\widetilde{z}_{s-1}^{-1}\varphi(z_{s-1}^{-1}a_{s,\widetilde{\pi}_s(\pi_s(r_s))}v)\widetilde{v})))\\
=\ \widetilde{\varphi}(\varphi((&\\
&(\varphi^{-1}(\widetilde{u})ua_{1,\widetilde{\pi}_1(\pi_1(1))}z_1\varphi^{-1}(\widetilde{z}_1),\dots,\varphi^{-1}(\widetilde{u})ua_{1,\widetilde{\pi}_1(\pi_1(r_1))}z_1\varphi^{-1}(\widetilde{z}_1)),\\
&(\varphi^{-1}(\widetilde{z}_1^{-1})z_1^{-1}a_{2,\widetilde{\pi}_2(\pi_2(1))}z_2\varphi^{-1}(\widetilde{z}_2),\dots,\varphi^{-1}(\widetilde{z}_1^{-1})z_1^{-1}a_{2,\widetilde{\pi}_2(\pi_2(r_2))}z_2\varphi^{-1}(\widetilde{z}_2)),\\
&\vdots\\
&(\varphi^{-1}(\widetilde{z}_{s-2}^{-1})z_{s-2}^{-1}a_{s-1,\widetilde{\pi}_{s-1}(\pi_{s-1}(1))}z_{s-1}\varphi^{-1}(\widetilde{z}_{s-1}),\dots,\\
&\varphi^{-1}(\widetilde{z}_{s-2}^{-1})z_{s-2}^{-1}a_{s-1,\widetilde{\pi}_{s-1}(\pi_{s-1}(r_{s-1}))}z_{s-1}\varphi^{-1}(\widetilde{z}_{s-1})),\\
&(\varphi^{-1}(\widetilde{z}_{s-1}^{-1})z_{s-1}^{-1}a_{s,\widetilde{\pi}_s(\pi_s(1))}v\varphi^{-1}(\widetilde{v}),\dots,\varphi^{-1}(\widetilde{z}_{s-1}^{-1})z_{s-1}^{-1}a_{s,\widetilde{\pi}_s(\pi_s(r_s))}v\varphi^{-1}(\widetilde{v}))))))\\
=\ &\underbrace{(\mathfrak{a}_{\widetilde{\varphi}\circ\varphi}\circ\mathfrak{e}_{(\widetilde{\pi}_1\circ\pi_1,\dots,\widetilde{\pi}_s\circ\pi_s)}\circ\mathfrak{s}_{(z_1\varphi^{-1}(\widetilde{z}_1),\dots,z_{s-1}\varphi^{-1}(\widetilde{z}_{s-1}))}\circ\mathfrak{t}_{\varphi^{-1}(\widetilde{u})u,v\varphi^{-1}(\widetilde{v})})}_{\in\mathfrak{M}_{(r_1,\dots,r_s)}(G)}(\alpha).
\end{aligned}
$$

So, the two transformations from $\mathfrak{M}_{(r_1,\dots,r_s)}(G)$ can be combined to a single transformation in $\mathfrak{M}_{(r_1,\dots,r_s)}(G)$, namely a translation by $\varphi^{-1}(\widetilde{u})u \in G$ from the left and $v\varphi^{-1}(\widetilde{v}) \in G$ from the right, a sandwich using the elements $(z_1\varphi^{-1}(\widetilde{z}_1),\dots,z_{s-1}\varphi^{-1}(\widetilde{z}_{s-1})) \in G^{s-1}$, an element shuffle using the permutations $(\widetilde{\pi}_1\circ\pi_1,\dots,\widetilde{\pi}_s\circ\pi_s) \in \mathrm{Sym}(r_1)\times\ldots\times\mathrm{Sym}(r_s)$ and an application of the automorphism $\widetilde{\varphi}\circ\varphi \in \mathrm{Aut}(G)$.

The identity in $\mathfrak{M}_{(r_1,\dots,r_s)}(G)$ is $\mathfrak{a}_{\mathrm{id}}\circ\mathfrak{e}_{(\mathrm{id},\dots,\mathrm{id})}\circ\mathfrak{s}_{(\mathrm{id},\dots,\mathrm{id})}\circ\mathfrak{t}_{\mathrm{id},\mathrm{id}}$.

The inverse of a $\mathfrak{a}_\varphi\circ\mathfrak{e}_{(\pi_1,\dots,\pi_s)}\circ\mathfrak{s}_{(z_1,\dots,z_{s-1})}\circ\mathfrak{t}_{u,v} \in \mathfrak{M}_{(r_1,\dots,r_s)}(G)$ is

$$
\begin{aligned}
&(\mathfrak{a}_\varphi\circ\mathfrak{e}_{(\pi_1,\dots,\pi_s)}\circ\mathfrak{s}_{(z_1,\dots,z_{s-1})}\circ\mathfrak{t}_{u,v})^{-1}\\
=\ &\mathfrak{t}_{u,v}^{-1}\circ\mathfrak{s}_{(z_1,\dots,z_{s-1})}^{-1}\circ\mathfrak{e}_{(\pi_1,\dots,\pi_s)}^{-1}\circ\mathfrak{a}_\varphi^{-1}\\
=\ &\mathfrak{t}_{u^{-1},v^{-1}}\circ\mathfrak{s}_{(z_1^{-1},\dots,z_{s-1}^{-1})}\circ\mathfrak{e}_{(\pi_1^{-1},\dots,\pi_s^{-1})}\circ\mathfrak{a}_{\varphi^{-1}}\\
=\ &\underbrace{\mathfrak{a}_{\varphi^{-1}}\circ\mathfrak{e}_{(\pi_1^{-1},\dots,\pi_s^{-1})}\circ\mathfrak{s}_{(\varphi(z_1^{-1}),\dots,\varphi(z_{s-1}^{-1}))}\circ\mathfrak{t}_{\varphi(u^{-1}),\varphi(v^{-1})}}_{\in\mathfrak{M}_{(r_1,\dots,r_s)}(G)}.
\end{aligned}
$$

So, $\mathfrak{M}_{(r_1,\dots,r_s)}(G)$ is a group. □

Remark 5.19. Let $f_0(U) = \{\mathrm{id}\}$ and $f_1(U) = U$. As it can be seen from the proof of Proposition 5.18, the sets $\mathfrak{T}(G)$, $\mathfrak{S}_s(G)$, $\mathfrak{E}_{(r_1,\dots,r_s)}(G)$ and $\mathfrak{A}(G)$ are independent in the sense that

$$f_{b_4}(\mathfrak{A}(G)) \circ f_{b_3}(\mathfrak{E}_{(r_1,\dots,r_s)}(G)) \circ f_{b_2}(\mathfrak{S}_s(G)) \circ f_{b_1}(\mathfrak{T}(G)) \leq \mathfrak{M}_{(r_1,\dots,r_s)}(G)$$

for all $b_1, b_2, b_3, b_4 \in \mathbb{Z}_2$.

Furthermore, let $T \leq \mathfrak{T}(G)$, $S \leq \mathfrak{S}_s(G)$, $E \leq \mathfrak{E}_{(r_1,\dots,r_s)}(G)$ and $A \leq \mathfrak{A}(G)$, then

$$A \circ E \circ f_{b_2}(\mathfrak{S}_s(G)) \circ f_{b_1}(\mathfrak{T}(G)) \leq \mathfrak{M}_{(r_1,\dots,r_s)}(G),$$
$$E \circ S \circ T \leq \mathfrak{M}_{(r_1,\dots,r_s)}(G)$$

for all $b_1, b_2 \in \mathbb{Z}_2$.

Remark 5.20. Let $\mathfrak{H} \leq \mathfrak{M}_{(r_1,\dots,r_s)}(G)$. Then an equivalence relation $\sim_{\mathfrak{H}}$ can be defined, where $\alpha \sim_{\mathfrak{H}} \beta$ holds if and only if $t(\alpha) = t(\beta) = (r_1, \dots, r_s)$ and there exists a transformation in $\mathfrak{H}$ that maps α to β (or the other way around).

5.2. Irreducibility

Let G be a group. We call a set (or block containing no duplicate elements) $A \subseteq G$ *irreducible*, if there are no sets (or blocks) $B, C \subseteq G$ with $|B| > 1$, $|C| > 1$, $|A| = |B|\,|C|$ and $A = BC$.

For finding special conditions for irreducibility, we have a look at homomorphisms to smaller groups.

Let $\varphi : G \to H$ be a group homomorphism. For a set or multiset $A \subseteq G$, we write $\varphi(A)$ for the (unordered) multiset $[\varphi(a) \mid a \in A]$.

Proposition 5.21. *Let G and H be groups, $\varphi : G \to H$ a group homomorphism, and $A, B, C \subseteq G$ with $|A| = |B|\,|C|$. Let $A' := \varphi(A), B' := \varphi(B), C' := \varphi(C)$.*

A necessary condition for $A = BC$ is $A' = B'C'$.

5.2.1. Conditions Induced by Group Homomorphisms to $\mathbb{Z}_2$

In this section, we derive a few irreducibility conditions based on counting images of group homomorphisms to $\mathbb{Z}_2$.

In the whole section, let $\varphi : G \to \mathbb{Z}_2$ be a group homomorphism.

Example 5.22. Let G be a group, $A \subseteq G$, and $A' := \varphi(A)$ (multiset). If $\mathfrak{c}_0(A') = 1$ or $\mathfrak{c}_1(A') = 1$, then A is irreducible.

Proof. Let us assume there exist $B, C \subseteq G$ with $|B| > 1$, $|C| > 1$, $|A| = |B|\,|C|$ and $A = BC$. Let $B' := \varphi(B)$ and $C' := \varphi(C)$. One of the following cases must occur:

$\mathfrak{c}_0(B')$	$\mathfrak{c}_1(B')$	$\mathfrak{c}_0(C')$	$\mathfrak{c}_1(C')$	$\mathfrak{c}_0(A')$	$\mathfrak{c}_1(A')$
0	$\lvert B\rvert$	0	$\lvert C\rvert$	$\lvert A\rvert$	0
0	$\lvert B\rvert$	$\lvert C\rvert$	0	0	$\lvert A\rvert$
$\lvert B\rvert$	0	0	$\lvert C\rvert$	0	$\lvert A\rvert$
$\lvert B\rvert$	0	$\lvert C\rvert$	0	$\lvert A\rvert$	0
$*$	$*$	>0	>0	≥ 2	≥ 2
>0	>0	$*$	$*$	≥ 2	≥ 2

In all cases, $\mathfrak{c}_0(A') \neq 1$ and $\mathfrak{c}_1(A') \neq 1$. So, if $\mathfrak{c}_0(A') = 1$ or $\mathfrak{c}_1(A') = 1$, A is irreducible. □

Proposition 5.23. *Let G be a group and $A, B, C \subseteq G$, such that $|A| = |B|\,|C|$ and $|B| > 1$, $|C| > 1$. We write $l := |A|, r := |B|$ and $s := |C|$, $A' := \varphi(A)$, $B' := \varphi(B)$, $C' := \varphi(C)$.*

The following is a necessary condition for $A = BC$:

$$\mathfrak{c}_0(A') = l + 2 \cdot \mathfrak{c}_0(B') \cdot \mathfrak{c}_0(C') - s \cdot \mathfrak{c}_0(B') - r \cdot \mathfrak{c}_0(C'),$$

and equivalently

$$\mathfrak{c}_1(A') = -2 \cdot \mathfrak{c}_1(B') \cdot \mathfrak{c}_1(C') + s \cdot \mathfrak{c}_1(B') + r \cdot \mathfrak{c}_1(C').$$

Proof. Let us assume $A = BC$. 0 can be generated in $B' + C'$ by $b'_i + c'_j$ with $b'_i \in B'$, $c'_j \in C'$ and either $b'_i = c'_j = 0$ or $b'_i = c'_j = 1$ (because $1 + 1 = 0$ in $\mathbb{Z}_2$). So, we have

$$\begin{aligned}
\mathfrak{c}_0(A') &= \mathfrak{c}_0(B')\mathfrak{c}_0(C') + \mathfrak{c}_1(B')\mathfrak{c}_1(C') \\
&= \mathfrak{c}_0(B')\mathfrak{c}_0(C') + (r - \mathfrak{c}_0(B'))(s - \mathfrak{c}_0(C')) \\
&= \mathfrak{c}_0(B')\mathfrak{c}_0(C') + rs - s\mathfrak{c}_0(B') - r\mathfrak{c}_0(C') + \mathfrak{c}_0(B')\mathfrak{c}_0(C') \\
&= l + 2\mathfrak{c}_0(B')\mathfrak{c}_0(C') - s\mathfrak{c}_0(B') - r\mathfrak{c}_0(C').
\end{aligned}$$

And thus

$$\begin{aligned}
l - \mathfrak{c}_1(A') &= l + 2(r - \mathfrak{c}_1(B'))(s - \mathfrak{c}_1(C')) - s(r - \mathfrak{c}_1(B')) - r(s - \mathfrak{c}_1(C')) \\
\Rightarrow -\mathfrak{c}_1(A') &= 2(rs - r\mathfrak{c}_1(C') - s\mathfrak{c}_1(B') + \mathfrak{c}_1(B')\mathfrak{c}_1(C')) - rs + s\mathfrak{c}_1(B') - rs + r\mathfrak{c}_1(C') \\
\Rightarrow \mathfrak{c}_1(A') &= 2(r\mathfrak{c}_1(C') + s\mathfrak{c}_1(B') - \mathfrak{c}_1(B')\mathfrak{c}_1(C')) - s\mathfrak{c}_1(B') - r\mathfrak{c}_1(C') \\
&= -2\mathfrak{c}_1(B')\mathfrak{c}_1(C') + s\mathfrak{c}_1(B') + r\mathfrak{c}_1(C').
\end{aligned}$$

□

Example 5.24. Let $A \subseteq G$ with $|A| = 2^m$ for some $m \in \mathbb{N}$, and $A' := \varphi(A)$. If $\mathfrak{c}_0(A')$ is odd, then A is irreducible.

Proof. Let us assume $A = BC$. As $l = |A| = 2^m$, there must exist $u, v \in \mathbb{N}$ such that $r = |B| = 2^u$ and $s = |C| = 2^v$ (and $u + v = m$). So, especially r and s are even. Consequently the right side of the formula in Proposition 5.23 is even (because l, 2, s and r are even). If $\mathfrak{c}_0(A')$ is odd, the formula cannot be true, i.e. such B and C do not exist. □

Example 5.25. Let $A \subseteq G$ with $|A| = p^m$ for some $p \in \mathbb{P}$ and $m \in \mathbb{N}$. Write $A' := \varphi(A)$, $B' := \varphi(B)$ and $C' := \varphi(C)$.

$A = BC$ can only be true, if $\mathfrak{c}_0(A') \equiv 2 \cdot \mathfrak{c}_0(B') \cdot \mathfrak{c}_0(C') \pmod{p}$.

Proof. Let us assume $A = BC$. As $l = |A| = p^m$, there must exist $u, v \in \mathbb{N}$ such that $r = |B| = p^u$ and $s = |C| = p^v$ (and $u + v = m$). By reducing both sides of the formula mod p, we obtain:

$$\begin{aligned} & \mathfrak{c}_0(A') = l + 2\mathfrak{c}_0(B')\mathfrak{c}_0(C') - s\mathfrak{c}_0(B') - r\mathfrak{c}_0(C') \\ \Rightarrow\ & \mathfrak{c}_0(A') \equiv p^m + 2\mathfrak{c}_0(B')\mathfrak{c}_0(C') - p^v\mathfrak{c}_0(B') - p^u\mathfrak{c}_0(C') \pmod{p} \\ \Rightarrow\ & \mathfrak{c}_0(A') \equiv 2\mathfrak{c}_0(B')\mathfrak{c}_0(C') \pmod{p}. \end{aligned}$$

□

5.2.2. Group, Block and Homomorphism Dependency

Depending on the group and the block, different homomorphisms may provide different amounts of information on the irreducibility of the block.

Example 5.26. Let

$$\vartheta : D_{2n} \to (\mathbb{Z}_2, +) : g \mapsto \begin{cases} 0, & \text{if } g \text{ is a rotation,} \\ 1, & \text{if } g \text{ is a reflection,} \end{cases}$$

$$\varpi : G \to (\mathbb{Z}_2, +) : g \mapsto \begin{cases} 0, & \text{if } \operatorname{sgn}(g) = 1, \\ 1, & \text{if } \operatorname{sgn}(g) = -1, \end{cases}$$

where $\operatorname{sgn}(g)$ is the sign of the permutation g, when G is represented as a permutation group. For a permutation representation of D_{2n}, see Section 2.3.3.3.

Clearly, both ϑ and ϖ are group homomorphisms.

The following examples show that sometimes ϑ allows to decide that a block is irreducible and sometimes ϖ allows this decision:

- Let $G = D_{2\cdot 5}$ and $A = \{\mathrm{id}, \sigma^2, \sigma^3, \sigma^2\tau\}$. Proposition 5.23 with ϖ as homomorphism does not tell us anything about the irreducibility of A, because $\operatorname{sgn}(g) = 1$ for all $g \in G$ and the formula is fulfilled by $4 = 4 + 2 \cdot 2 \cdot 2 - 2 \cdot 2 - 2 \cdot 2$. However, using Proposition 5.23 with ϑ as homomorphism we can conclude that A is irreducible (we have $l = 4$, $r = s = 2$, thus the right side of the formula is even, however $\mathscr{c}_\sigma(A) = 3$, i.e. the formula cannot be fulfilled and consequently A is irreducible).

- Let $G = D_{2\cdot 6}$ and $A = \{\mathrm{id}, \sigma, \sigma^2, \sigma^4\}$. Proposition 5.23 with ϑ as homomorphism does not tell us anything about the irreducibility of A, because $\mathscr{c}_\sigma(A) = 4$ and the formula could be fulfilled, e.g. by $4 = 4 + 2\cdot 2\cdot 2 - 2\cdot 2 - 2\cdot 2$ or $4 = 4 + 2\cdot 0\cdot 0 - 2\cdot 0 - 2\cdot 0$. However, using Proposition 5.23 with ϖ as homomorphism we can conclude that A is irreducible (we have $l = 4$, $r = s = 2$, thus the right side of the formula is even, however $\operatorname{sgn}(\mathrm{id}) = \operatorname{sgn}(\sigma^2) = \operatorname{sgn}(\sigma^4) = 1$, $\operatorname{sgn}(\sigma) = -1$, so $\mathsf{c}_e(A) = 3$, i.e. the formula cannot be fulfilled and consequently A is irreducible).

- Let $G = D_{2\cdot 7}$. In this group, for all $g \in G$ we have $\operatorname{sgn}(g) = 1 \Leftrightarrow g$ is a rotation (see Section 2.3.3.3), so ϑ and ϖ provide exactly the same information; $\mathsf{c}_e(M) = \mathscr{c}_\sigma(M)$ for all $M \subseteq G$.

5.3. Linear Representations and Characters

Let $\rho : G \to \mathrm{GL}(n, \mathbb{C})$ a $\mathbb{C}$-representation, and χ a character.

For a subset or multiset $A \subseteq G$, let $\acute{\rho}(A) := \sum_{g \in A} \rho(g)$. Similarly, define $\acute{\chi}(A) := \sum_{g \in A} \chi(g)$.

One of the most important tools for studying factorizations of abelian groups using characters in [Sza04] is the following: let G be an abelian group and $A, B \subseteq G$, then $A = B$ if and only if $\acute{\chi}(A) = \acute{\chi}(B)$ for all characters χ of G.

We now generalize this for non-abelian groups (and prove it using a similar approach as in [Sza04]).

Theorem 5.27. *Let G be a group and $A, B \subseteq G$. $\mathrm{Cl}(A) = \mathrm{Cl}(B)$ if and only if $\acute{\chi}(A) = \acute{\chi}(B)$ for all characters χ of G.*

Proof. Let $c_1, \ldots, c_m \in G$ be representatives for the conjugacy classes of G (i.e. $\bigcup_{i=1}^{m} \mathrm{Cl}(c_i) = G$ and $\mathrm{Cl}(c_i) \cap \mathrm{Cl}(c_j) = \emptyset$ for all $i \neq j$). For every character χ of G, we have

$$\acute{\chi}(A) = \sum_{g \in A} \chi(g) = \sum_{i=1}^{m} |\mathrm{Cl}(c_i) \cap A| \, \chi(c_i)$$

and thus

$$\begin{aligned}
\acute{\chi}(A) = \acute{\chi}(B) &\Leftrightarrow \sum_{i=1}^{m} |\mathrm{Cl}(c_i) \cap A| \, \chi(c_i) = \sum_{i=1}^{m} |\mathrm{Cl}(c_i) \cap B| \, \chi(c_i) \\
&\Leftrightarrow \sum_{i=1}^{m} |\mathrm{Cl}(c_i) \cap A| \, \chi(c_i) - \sum_{i=1}^{m} |\mathrm{Cl}(c_i) \cap B| \, \chi(c_i) = 0 \\
&\Leftrightarrow \sum_{i=1}^{m} \left(|\mathrm{Cl}(c_i) \cap A| - |\mathrm{Cl}(c_i) \cap B| \right) \chi(c_i) = 0.
\end{aligned}$$

There are exactly as many conjugacy classes as characters, i.e. there are m characters $\chi_1, \ldots, \chi_m$. The above holds for all characters if and only if

$$\underbrace{\begin{pmatrix} \chi_1(c_1) & \cdots & \chi_1(c_m) \\ \vdots & \ddots & \vdots \\ \chi_m(c_1) & \cdots & \chi_m(c_m) \end{pmatrix}}_{M} \cdot \underbrace{\begin{pmatrix} |\mathrm{Cl}(c_1) \cap A| - |\mathrm{Cl}(c_1) \cap B| \\ \vdots \\ |\mathrm{Cl}(c_m) \cap A| - |\mathrm{Cl}(c_m) \cap B| \end{pmatrix}}_{v} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Observe that M is the character table of G. Thus the columns are orthogonal. A set of pairwise orthogonal vectors is linear independent, so $\operatorname{rank}(M) = m$. Therefore, the linear

equation system $M \cdot v = (0, \ldots, 0)^t$ has exactly one solution: $v = (0, \ldots, 0)^t \in \mathbb{C}^m$. We get

$$\begin{aligned} & |\mathrm{Cl}(c_i) \cap A| - |\mathrm{Cl}(c_i) \cap B| = 0 \text{ for all } 1 \leq i \leq m \\ \Leftrightarrow\ & |\mathrm{Cl}(c_i) \cap A| = |\mathrm{Cl}(c_i) \cap B| \text{ for all } 1 \leq i \leq m \\ \Leftrightarrow\ & \mathrm{Cl}(A) = \mathrm{Cl}(B). \end{aligned}$$

□

Corollary 5.28. *Let G be an abelian group and $A, B \subseteq G$. $A = B$ if and only if $\acute{\chi}(A) = \acute{\chi}(B)$ for all characters χ of G.*

Proof. This follows from Theorem 5.27. As G is abelian, there are $|G|$ different conjugacy classes; we have $\mathrm{Cl}(g) = \{g\}$ for all $g \in G$. Thus, $\mathrm{Cl}(A) = \mathrm{Cl}(B) \Leftrightarrow A = B$. □

Example 5.29. Let $G = D_{2\cdot 3}$. From Section 2.3.3.2 we know the conjugacy classes: $\{\mathrm{id}\}$, $\{\sigma, \sigma^2\}$, $\{\tau, \sigma\tau, \sigma^2\tau\}$. Using the definitions in Section 2.3.3.4, there are three representations ρ_0, ρ_{-1} (of degree 1) and ρ_1 (of degree 2) with the following characters:

$$\begin{aligned} \chi_{\rho_0} &: G \to \mathbb{C} : \sigma^k\tau^c \mapsto 1, \\ \chi_{\rho_{-1}} &: G \to \mathbb{C} : \sigma^k\tau^c \mapsto (-1)^c, \\ \chi_{\rho_1} &: G \to \mathbb{C} : \sigma^k\tau^c \mapsto \begin{cases} \zeta_3^k + \zeta_3^{-k}, & \text{if } c = 0, \\ 0, & \text{if } c = 1. \end{cases} \end{aligned}$$

So, we get the character table for $G = D_{2\cdot 3}$ shown in Table 5.1 (note that $\zeta_3 + \zeta_3^2 = -1$).

$D_{2\cdot 3}$	id	σ	τ
χ_{ρ_0}	1	1	1
$\chi_{\rho_{-1}}$	1	1	-1
χ_{ρ_1}	2	-1	0

Table 5.1.: Character Table for $D_{2\cdot 3}$

Now, let $A := \{\mathrm{id}, \sigma, \sigma\tau, \sigma^2\tau\}$ and $B := \{\mathrm{id}, \sigma^2, \tau, \sigma\tau\}$. Then

$$\begin{aligned} \acute{\chi}_{\rho_0}(A) &= \chi_{\rho_0}(\mathrm{id}) + \chi_{\rho_0}(\sigma) + \chi_{\rho_0}(\sigma\tau) + \chi_{\rho_0}(\sigma^2\tau) = 1 + 1 + 1 + 1 = 4, \\ \acute{\chi}_{\rho_0}(B) &= \chi_{\rho_0}(\mathrm{id}) + \chi_{\rho_0}(\sigma^2) + \chi_{\rho_0}(\tau) + \chi_{\rho_0}(\sigma\tau) = 1 + 1 + 1 + 1 = 4, \\ \acute{\chi}_{\rho_{-1}}(A) &= \chi_{\rho_{-1}}(\mathrm{id}) + \chi_{\rho_{-1}}(\sigma) + \chi_{\rho_{-1}}(\sigma\tau) + \chi_{\rho_{-1}}(\sigma^2\tau) = 1 + 1 - 1 - 1 = 0, \\ \acute{\chi}_{\rho_{-1}}(B) &= \chi_{\rho_{-1}}(\mathrm{id}) + \chi_{\rho_{-1}}(\sigma^2) + \chi_{\rho_{-1}}(\tau) + \chi_{\rho_{-1}}(\sigma\tau) = 1 + 1 - 1 - 1 = 0, \\ \acute{\chi}_{\rho_1}(A) &= \chi_{\rho_1}(\mathrm{id}) + \chi_{\rho_1}(\sigma) + \chi_{\rho_1}(\sigma\tau) + \chi_{\rho_1}(\sigma^2\tau) = 2 - 1 + 0 + 0 = 1, \\ \acute{\chi}_{\rho_1}(B) &= \chi_{\rho_1}(\mathrm{id}) + \chi_{\rho_1}(\sigma^2) + \chi_{\rho_1}(\tau) + \chi_{\rho_1}(\sigma\tau) = 2 - 1 + 0 + 0 = 1. \end{aligned}$$

So we have $\acute{\chi}(A) = \acute{\chi}(B)$ for all characters χ of G and can easily verify that indeed $\mathrm{Cl}(A) = [\mathrm{Cl}(\mathrm{id}), \mathrm{Cl}(\sigma), \mathrm{Cl}(\tau), \mathrm{Cl}(\tau)] = \mathrm{Cl}(B)$, i.e. the multisets of conjugacy classes are the same, but $A \neq B$.

Theorem 5.30. *Let G be a group and $A, B \subseteq G$. The following conditions are necessary (but not sufficient) for $A = B$:*

- *$\acute{\chi}(A) = \acute{\chi}(B)$ for every character χ of G.*
- *$\acute{\chi}(A) = \acute{\chi}(B)$ for every character χ of U, where $U \leq G$ is a subgroup of G for which additionally $A \subseteq U$ and $B \subseteq U$ are true.*

Example 5.31. Let $G = D_{2\cdot 3}$, $A := \{\mathrm{id}, \sigma\}$ and $B := \{\mathrm{id}, \sigma^2\}$. Using the character table of G in Example 5.29, we easily compute $\acute{\chi}_{\rho_0}(A) = 2 = \acute{\chi}_{\rho_0}(B)$, $\acute{\chi}_{\rho_{-1}}(A) = 2 = \acute{\chi}_{\rho_{-1}}(B)$, $\acute{\chi}_{\rho_1}(A) = 1 = \acute{\chi}_{\rho_1}(B)$, i.e. the characters of G do not allow us to conclude $A \neq B$.

$U := \{\mathrm{id}, \sigma, \sigma^2\} \cong \mathbb{Z}_3$ is a subgroup of G. It is cyclic, abelian, and thus every representation is of degree 1 and each element forms an own conjugacy class. Let $\zeta_3 = e^{\frac{2\pi i}{3}}$. The generator element can be mapped to any power of ζ_3, so we easily compute the character table, shown in Table 5.2.

$\mathbb{Z}_3$	id	σ	σ^2
χ_{ρ_0}	1	1	1
χ_{ρ_1}	1	ζ_3	ζ_3^2
χ_{ρ_2}	1	ζ_3^2	ζ_3

Table 5.2.: Character Table for $\mathbb{Z}_3$

Now, observe that $A, B \subseteq U$. We for example see

$$\acute{\chi}_{\rho_1}(A) = 1 + \zeta_3 \neq 1 + \zeta_3^2 = \acute{\chi}_{\rho_1}(B)$$

and thus can conclude $A \neq B$ by Theorem 5.30.

Example 5.32. The conditions in Theorem 5.30 are not sufficient for $A = B$.

For example, let $G = D_{2\cdot 3}$, $A = \{\mathrm{id}, \tau\}$, $B = \{\mathrm{id}, \sigma\tau\}$. Then $\acute{\chi}(A) = \acute{\chi}(B)$ for all characters χ of G (obviously, because τ and $\sigma\tau$ are in the same conjugacy class).

Let U be the subgroup generated by $\{\mathrm{id}, \tau, \sigma\tau\}$, then U also contains $\tau \cdot \sigma\tau = \sigma^{-1} = \sigma^2$, i.e. $|U| \geq 4$. As $|U| \mid |G|$ by Lagrange's theorem, we get $|U| = 6$, and thus $U = G$. So, the subgroup character condition does not provide any additional information on whether $A = B$ or not.

Example 5.33. The conditions in Theorem 5.30 are not sufficient for replacing blocks in a logarithmic signature.

For example, let $G = D_{2\cdot 3}$, $A = \{\mathrm{id}, \tau\}$, $A' = \{\mathrm{id}, \sigma\tau\}$, $B = \{\mathrm{id}, \sigma\tau, \sigma\}$. Like previously, $\acute{\chi}(A) = \acute{\chi}(A')$ for all characters χ of G and its subgroups containing A and A'. Here, (A, B) is a logarithmic signature for G, but (A', B) is not.

5.4. Irreducibility and Characters

Lemma 5.34. *Let $\chi : G \to \mathbb{C}$ be a character of an abelian group G, then $\chi(g+h) = \chi(g)\chi(h)$ for all $g, h \in G$.*

This is not necessarily true for non-abelian groups.

Proof. Let G be abelian. From character theory we know that every complex, irreducible representation ρ is of degree 1, i.e. $\rho : G \to \mathbb{C}^\times$ (we have $\mathrm{GL}(1,\mathbb{C}) \cong \mathbb{C}^\times$) and thus $\chi_\rho(g) = \mathrm{tr}(\rho(g)) = \rho(g)$. As ρ is a group homomorphism, so is χ_ρ, i.e. $\chi(g+h) = \chi(g)\chi(h)$ for all $g, h \in G$.

We give a counter-example for the non-abelian case. Let $G = D_{2\cdot 3}$. For the character χ_{ρ_1} (see Example 5.29), we get

$$\chi_{\rho_1}(\mathrm{id} \cdot \mathrm{id}) = \chi_{\rho_1}(\mathrm{id}) = 2 \neq 4 = 2 \cdot 2 = \chi_{\rho_1}(\mathrm{id})\chi_{\rho_1}(\mathrm{id}). \qquad \square$$

Lemma 5.35. *Let $\chi : G \to \mathbb{C}$ be a character of an abelian group G, and $A, B \subseteq G$ multisets of G. Then $\acute{\chi}(A+B) = \acute{\chi}(A)\acute{\chi}(B)$.*

This is not necessarily true for non-abelian groups.

Proof. Let G be abelian, then χ is a group homomorphism according to Lemma 5.34. Note that $A + B$ is a multiset.

$$\acute{\chi}(A+B) = \sum_{a+b \in A+B} \chi(a+b) \overset{\text{hom.}}{=} \sum_{a+b \in A+B} \chi(a)\chi(b) = \sum_{a \in A} \chi(a) \cdot \sum_{b \in B} \chi(b) = \acute{\chi}(A)\acute{\chi}(B).$$

If G is non-abelian, this does not hold. For example, take $G = D_{2\cdot 3}$, $A = \{\mathrm{id}\}$ and $B = \{\mathrm{id}\}$.

$$\acute{\chi}_{\rho_1}(AB) = \acute{\chi}_{\rho_1}(\{\mathrm{id}\}) = \chi_{\rho_1}(\mathrm{id}) = 2 \neq 4 = 2 \cdot 2 = \chi_{\rho_1}(\mathrm{id})\chi_{\rho_1}(\mathrm{id}) = \acute{\chi}_{\rho_1}(A)\acute{\chi}_{\rho_1}(B). \qquad \square$$

Theorem 5.36. *Let G be an abelian group and $A, B, C \subseteq G$. Then $A = B + C$ if and only if $\acute{\chi}(A) = \acute{\chi}(B)\acute{\chi}(C)$ for all characters χ of G.*

If G is non-abelian, $\acute{\chi}(A) = \acute{\chi}(B)\acute{\chi}(C)$ does not even need to be true when $A = BC$.

Proof. Let G be abelian.

$$\begin{aligned} & A = B + C \\ \overset{\text{Cor. 5.28}}{\iff} \; & \acute{\chi}(A) = \acute{\chi}(B+C) \\ \overset{\text{Lem. 5.35}}{\iff} \; & \acute{\chi}(A) = \acute{\chi}(B)\acute{\chi}(C) \end{aligned}$$

for all characters χ of G.

A counter-example for the non-abelian case can be found in the proof of Lemma 5.35. $\square$

Proposition 5.37. *Let G be a group, $\rho : G \to \mathrm{GL}(n, K)$ a K-representation of G, and $A, B, C \subseteq G$.*

$\acute{\rho}(A) = \acute{\rho}(B)\acute{\rho}(C)$ is a necessary condition for $A = BC$.

If ρ is faithful, then $A = BC \Leftrightarrow \acute{\rho}(A) = \acute{\rho}(B)\acute{\rho}(C)$.

Proof. For every K-representation ρ of G we have:

$$\acute{\rho}(BC) = \sum_{bc \in BC} \rho(bc) \overset{\text{hom.}}{=} \sum_{bc \in BC} \rho(b)\rho(c) \overset{\text{ring}}{=} \sum_{b \in B} \rho(b) \cdot \sum_{c \in C} \rho(c) = \acute{\rho}(B)\acute{\rho}(C).$$

With this, it follows:

$$A = BC \overset{*}{\Rightarrow} \acute{\rho}(A) = \acute{\rho}(BC) \Leftrightarrow \acute{\rho}(A) = \acute{\rho}(B)\acute{\rho}(C).$$

(*) The direction to the left only follows if ρ is faithful. □

6. Generating Logarithmic Signatures

For most cryptographic primitives that use logarithmic signatures, we need to be able to generate random logarithmic signatures for a given group G.

There is interest in generating tame logarithmic signatures as well as wild ones. In cryptographic primitives, tame logarithmic signatures are for instance used for ordering (mapping a group element to an integer in $[0, |G|-1]$) or as keys for a symmetric encryption algorithm (like PGM in Section 4.1). Wild logarithmic signatures are required for instance in asymmetric encryption algorithms (like MST_1 in Section 4.2).

In the following, we analyze existing approaches and introduce new algorithms for generating logarithmic signatures.

One-way function/permutation. For an asymmetric encryption algorithm based on logarithmic signatures, the generation algorithm must provide information (the private key) that allows to efficiently compute element factorizations. If the algorithm does not provide this, it basically generates a logarithmic signature for being used as a one-way function/permutation, not a trap-door function/permutation. However, for a public-key system (like the generalized MST_1 system in Section 4.3), we need a trap-door function/permutation.

Algorithms for generating logarithmic signatures without providing knowledge how to compute element factorizations can be useful though for testing factorization algorithms. We therefore will describe such generation algorithms in the following, too.

Trap-door constructions. One possibility to construct logarithmic signatures is to start with a fixed tame logarithmic signature for a group G and randomly apply signature-preserving transformations (i.e. transformations that result in new logarithmic signatures) that allow deriving a factorization in the new one from the old one. Knowing the factorization of an element in the starting logarithmic signature, we can apply the transformations step-by-step and derive its factorization in the resulting logarithmic signature. The final logarithmic signature would be the public key, and the way how it was generated would be the private key.

Examples for this construction idea are amalgamated transversal logarithmic signatures (Section 6.3) and our new generation algorithm LS-Gen (Section 6.5).

An attacker trying to factorize an element would either try it directly (i.e. only using the public logarithmic signature) or try to find the way how the logarithmic signature was generated, i.e. try to find the applied transformations. The assumption/hope is that this is infeasible for the attacker.

Length. We are primarily interested in generating logarithmic signatures $\alpha \in \Lambda(G)$

where $\ell(\alpha)$ is polynomial in the minimal length $\min\{\ell(\beta) \mid \beta \in \Lambda(G)\}$. Larger logarithmic signatures are often trivial to generate (most simple example: all group elements stored in one block), but are not interesting due to space and complexity reasons (when using a logarithmic signature that requires space polynomial in $|G|$ to be stored/encoded, factoring using brute-force is efficient).

6.1. Exact Transversal Logarithmic Signatures

Collecting coset representatives for each of the subgroups in the next larger group in the subgroup chain $\gamma : \{\mathrm{id}\} = G_0 < G_1 < \ldots < G_{s-1} < G_s = G$ as blocks yields an exact transversal logarithmic signature (for details, see Section 3.4.3).

If $|G_i : G_{i-1}| \in \mathbb{P}$ for all $1 \leq i \leq s$, the resulting logarithmic signature has minimal length (by Remark 3.5).

6.2. Randomizing Elements

The most simple approach for generating a random, canonical logarithmic signature for a group G is the following:

1. Find numbers $m_1, \ldots, m_n \in \mathbb{N}_{\geq 2}$ such that $|G| = m_1 \cdot m_2 \cdots m_n$ (the m_i do not need to be prime).
2. For each m_i generate $m_i - 1$ random elements $g_{i,j} \in G$ and let
$$A_i := (\mathrm{id}, g_{i,1}, g_{i,2}, \ldots, g_{i,m_i-1}).$$
3. Test whether $\alpha := (A_1, A_2, \ldots, A_n)$ is a logarithmic signature for G. If yes, α is the result; if no, go to step 1.

Every canonical logarithmic signature (without blocks of size 1) may be generated using this approach.

Analysis. There is a multitude of issues with this approach:

- In the last step, it has to be tested whether α is a logarithmic signature for G. It is not known how to do this for large groups. For small groups G, one could simply compute all products and look whether there are any duplicates (if there is a duplicate, α is not a logarithmic signature for G). However, for larger groups this becomes infeasible.
- The algorithm does not provide a way to compute element factorizations, i.e. it is not usable in public-key systems.
- It usually is very inefficient. Of course this depends on G, but in general the algorithm will have to generate extremely many block sequences until finding a logarithmic signature. Some computer experiments are shown in Table 6.1.

Group	Log. Sig. Type	Rel. Freq.	Gen. Log. Sig.	Gen. Block Seq.
$D_{2\cdot 2}$	$(2,2)$	0.3750	41528868	110746276
$D_{2\cdot 3}$	$(2,3)$	0.1111	6150091	55337810
$D_{2\cdot 4}$	$(2,2,2)$	0.2188	15014920	68635483
$D_{2\cdot 5}$	$(2,5)$	0.0192	1379257	71889016
$D_{2\cdot 6} \cong D_{2\cdot 3} \times \mathbb{Z}_2$	$(2,2,3)$	0.0417	3135867	75263238
$D_{2\cdot 7}$	$(2,7)$	0.0031	143087	46413430
$D_{2\cdot 8}$	$(2,2,2,2)$	0.0879	3876003	44082541
$D_{2\cdot 9}$	$(2,3,3)$	0.0055	526478	95879030
	$(3,2,3)$	0.0055	457683	83515679
$D_{2\cdot 10}$	$(2,2,5)$	0.0036	119219	33176932
	$(2,5,2)$	0.0072	243903	33897095
$D_{2\cdot 15}$	$(2,3,5)$	0.00020	5603	27900811
	$(2,5,3)$	0.00016	4845	30460012
	$(3,2,5)$	0.00022	7419	33020317
$D_{2\cdot 2^4}$	$(2,2,2,2,2)$	0.0227	403467	17757624
$D_{2\cdot 2^5}$	$(2,\ldots,2)$	0.0036	48911	13497063
$D_{2\cdot 2^6}$	$(2,\ldots,2)$	0.00034	57607	168553491
$D_{2\cdot 2^7}$	$(2,\ldots,2)$	$1.9 \cdot 10^{-5}$	1195	62032198
$D_{2\cdot 2^8}$	$(2,\ldots,2)$	$6 \cdot 10^{-7}$	160	287727650
Sym(4)	$(2,3,4)$	0.00115	162232	141268071
	$(2,4,3)$	0.00140	269030	191853951
	$(3,2,4)$	0.00131	141943	108277507
	$(2,2,2,3)$	0.00753	2576706	342367457
	$(2,2,3,2)$	0.00543	575083	105895557
	$(2,3,2,2)$	0.00543	708234	130396918
	$(3,2,2,2)$	0.00753	1151281	152842645
Alt(4)	$(3,4)$	0.0058	573101	98935222
	$(2,2,3)$	0.0139	1348195	97029285
Alt(5)	$(2,2,3,5)$	$4.5 \cdot 10^{-6}$	1969	434941512
	$(2,3,2,5)$	$1.5 \cdot 10^{-6}$	603	406779226
	$(3,2,2,5)$	$6.0 \cdot 10^{-6}$	2209	367800602
	$(3,5,2,2)$	$3.8 \cdot 10^{-6}$	1422	375537742
$\mathbb{Z}_6$	$(2,3)$	0.0556	6846815	123245677
$\mathbb{Z}_{10}$	$(2,5)$	0.0048	1089460	227015422
$\mathbb{Z}_{12}$	$(2,2,3)$	0.0116	1329448	114811347
$\mathbb{Z}_3 \oplus \mathbb{Z}_2^2 \cong \mathbb{Z}_6 \oplus \mathbb{Z}_2$	$(2,2,3)$	0.0208	4425632	212544153
$\mathbb{Z}_{14}$	$(2,7)$	0.00048	146797	306889327
$\mathbb{Z}_{15}$	$(3,5)$	0.00044	59915	135452677
$\mathbb{Z}_{18}$	$(2,3,3)$	0.00114	204439	178895592
$\mathbb{Z}_{20}$	$(2,2,5)$	0.00051	109481	214704056
$\mathbb{Z}_{21}$	$(3,7)$	$3 \cdot 10^{-5}$	11644	393279710
$\mathbb{Z}_{22}$	$(2,11)$	$6 \cdot 10^{-6}$	1892	293677388

Group	Log. Sig. Type	Rel. Freq.	Gen. Log. Sig.	Gen. Block Seq.
$\mathbb{Z}_{24}$	$(2,2,2,3)$	0.00166	764118	460662026
$\mathbb{Z}_{30}$	$(2,3,5)$	$2.4 \cdot 10^{-5}$	17869	746020281
$\mathbb{Z}_{32}$	$(2,2,2,2,2)$	0.0037	432428	118130151
$\mathbb{Z}_{16} \oplus \mathbb{Z}_2$	$(2,2,2,2,2)$	0.0110	866709	78834511
$\mathbb{Z}_8 \oplus \mathbb{Z}_2^2$	$(2,2,2,2,2)$	0.0357	6779772	189721692
$\mathbb{Z}_4 \oplus \mathbb{Z}_2^3$	$(2,2,2,2,2)$	0.1154	16650187	144282242
$\mathbb{Z}_2^5$	$(2,2,2,2,2)$	0.2980	27819507	93361122

Table 6.1.: Relative Frequencies of Finding Logarithmic Signatures

We obtained the results in Table 6.1 by randomly generating canonical block sequences (not necessarily different) for the given group and type, and testing for each generated block sequence whether it is a logarithmic signature (by computing the set of all generated elements and testing whether this set is the whole group). The relative frequency (column "Rel. Freq." in the table) is the number of logarithmic signatures (column "Gen. Log. Sig.") divided by the number of generated block sequences (column "Gen. Block Seq."). We would like to emphasize again that we did not enforce the generated block sequences to be different, thus "Gen. Block Seq." is not the total number of different block sequences and "Gen. Log. Sig." is not the total number of different logarithmic signatures for the specified group and type.

The results were computed using the LogSig utility (Chapter 12), with the command line option "`-crf`". The group is passed using the parameter "`-g:`" and the type using "`-bstype:`". For improved performance, you might additionally want to specify "`-rand:fast`", which makes LogSig use a fast random number generator instead of a cryptographically secure one.

The program runs until being terminated manually, and approximately every 10 seconds it outputs the relative frequency, the number of logarithmic signatures and the number of generated block sequences.

For example, for $G = \mathrm{Sym}(4)$ and type $(2,2,3,2)$, the command line can look as follows:

```
LogSig.exe -crf -g:Sym4 -bstype:2,2,3,2
```

Interestingly, generating random covers with high probability is a lot easier, see [Sva07].

6.3. Amalgamated Transversal Logarithmic Signatures (Abelian Only)

A generation algorithm described in [Bla09] is the following.

Start with an exact transversal logarithmic signature α of an abelian group G and perform any of the following transformations a finite number of times:

- Permute the blocks of α.

- Permute the elements within a block of α.
- Replace a block A by a translate $A + g$ for some $g \in G$.
- Amalgamate: replace two blocks A_i and A_j by one (larger) block

$$A_i + A_j = \{g + h \mid g \in A_i, h \in A_j\}.$$

As G is abelian, each of these transformations results in another logarithmic signature for G, and thus the final output also is a logarithmic signature for G.

The set of logarithmic signatures generated by this algorithm is denoted by $\mathcal{AT}(G)$ (for Amalgamated Transversal).

In Section 8.9.1 we show that amalgamated transversal logarithmic signatures for $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ (with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$) are tame.

6.4. Aperiodic Logarithmic Signatures (Abelian Only)

Aperiodic logarithmic signatures are interesting, because periodicity often allows efficient factorization algorithms (e.g. see Algorithm 8.23).

6.4.1. p-Groups

We cite the following three lemmas from [Sza04].

Lemma 6.1. *Let H be a subgroup of the finite abelian group G, where $|G : H| = p$ is an odd prime. If $(q_1, \ldots, q_n)$ is not a periodicity forcing factorization type for H, then $(q_1, \ldots, q_n, p)$ and $(q_1, \ldots, q_{i-1}, q_i p, q_{i+1}, \ldots, q_n)$ are not periodicity forcing factorization types for G, for each i, $1 \leq i \leq n$.*

Proof. See [Sza04], Lemma 2.3.1. □

Lemma 6.2. *Let H be a subgroup of the finite cyclic group G, where $|G : H| = t \geq 2$. If $(r_1, \ldots, r_n)$ is not a periodicity forcing factorization type for H, then $(r_1, \ldots, r_n, t)$ and $(r_1, \ldots, r_{i-1}, r_i t, r_{i+1}, \ldots, r_n)$ are not periodicity forcing factorization types for G for each i, $1 \leq i \leq n$.*

Proof. See [Sza04], Lemma 6.3.1. □

Lemma 6.3. *Let G be a finite abelian 2-group which is not elementary abelian and let H be a subgroup of G for which $|G : H| = 2$. If $(q_1, \ldots, q_n)$ is not a periodicity forcing factorization type for H, then $(q_1, \ldots, q_n, 2)$ and $(q_1, \ldots, q_{i-1}, 2q_i, q_{i+1}, \ldots, q_n)$ are not periodicity forcing factorization types for G, for each i, $1 \leq i \leq n$.*

Proof. See [Sza04], Lemma 7.3.2. □

In the proofs, aperiodic factorizations for G are constructed explicitly from aperiodic factorizations of H.

We want to quickly outline the main idea used in the first construction. Let $(A_1, \ldots, A_n)$ be an aperiodic factorization of H in G. The factor group G/H is cyclic. Choose a generator element b (such that $\langle b + H \rangle = G/H$) and let $B := \{0, b, 2b, \ldots, (p-1)b\}$. If B is not a subgroup, we are done: $(A_1, \ldots, A_n, B)$ is an aperiodic factorization for G. If B is a subgroup, choose any $d \in H \setminus \{0\}$ and replace $(p-1)b$ by $(p-1)b + d$ in B (i.e. perform a selective shift on $(p-1)b$ by d), then $(A_1, \ldots, A_n, B)$ is an aperiodic factorization for G.

It can be shown that merging the new block with an existing one does not result in a periodic block.

These lemmas can of course be used to design algorithms for generating aperiodic logarithmic signatures.

We present one special generation algorithm. Given $p \in \mathbb{P}_{\geq 3}$ and $4 \leq n \in \mathbb{N}$, the algorithm outputs an abelian p-group G of order $|G| = p^n$ and an aperiodic logarithmic signature for G. The output group G has the property that its component orders are random; especially G will usually be far away from being cyclic or elementary abelian. In the output logarithmic signature, blocks are of size p^2 at most.

Algorithm 6.4. Let $p \in \mathbb{P}_{\geq 3}$ and $4 \leq n \in \mathbb{N}$.
Function `GenAperiodic`(Int p, Int n) : Group $\times$ LogSig

1. Set $C \leftarrow$ `new` List<Int>(). C.`Append`(p, p, p, p). Set $c \leftarrow 4$.
2. Set $\alpha \leftarrow$ aperiodic logarithmic signature for $\mathbb{Z}_p^4$, where α consists of two blocks of size p^2 each.
3. For $r \leftarrow 5$ to n:
   a) Let y be a random number out of $\{1, 2, \ldots, c+1\}$.
   b) If $y \leq c$:
      i. Set $C[y] \leftarrow C[y] \cdot p$.
      ii. Replace every element $(g_1, \ldots, g_c)$ in α by $(g_1, \ldots, g_{y-1}, g_y \cdot p, g_{y+1}, \ldots, g_c)$.

      else (i.e. if $y = c+1$):
      i. C.`Append`(p).
      ii. Replace every element $(g_1, \ldots, g_c)$ in α by $(g_1, \ldots, g_c, 0)$.
      iii. Set $c \leftarrow c + 1$.
   c) Generate c random numbers $(b_1, \ldots, b_c)$ with each $b_i \in \{0, 1, \ldots, C[i] - 1\}$.
   d) If $b_y \equiv 0 \pmod{p}$: set $b_y \leftarrow b_y +$ random number out of $\{1, 2, \ldots, p-1\}$.
   e) Let $b \leftarrow (b_1, \ldots, b_c)$ and set $B \leftarrow \{0, b, 2b, \ldots, (p-1)b\}$.
   f) While $B \leq G$:

i. Generate c random numbers $(d_1, \ldots, d_c)$ with each $d_i \in \{0, 1, \ldots, C[i]-1\}$. Set $d_y \leftarrow d_y \cdot p \bmod C[y]$. Let $d \leftarrow (d_1, \ldots, d_c)$.

ii. Set $B \leftarrow \{0, b, 2b, \ldots, (p-2)b, (p-1)b + d\}$.

g) Let l be the number of blocks in α and set $I \leftarrow \{i \mid 1 \leq i \leq l, |\alpha[i]| < p^2\}$.

h) Pick a random $s \in \mathbb{R}$ with $0 \leq s \leq 1$. Define $\eta := \frac{2}{3}$.

i) If $I = \emptyset$ or $s < \eta$: append B to α,
else: randomly pick an $x \in I$ and replace block $\alpha[x]$ by $\{a+b \mid a \in \alpha[x], b \in B\}$.

4. Let G be the abelian group with $t(G) = C$. `Return` (G, α).

Comments.

- The initial aperiodic logarithmic signature α for $\mathbb{Z}_p^4$ can be generated using brute-force or using the construction in Lemma 2.3.3 of [Sza04] (which is more efficient than brute-force).
- η controls the probability by which a new block (of size p) is added to α instead of extending an existing block (to size p^2). The larger η, the higher the probability that a new block is added.

Implementation. We have implemented this algorithm.

In order to generate a group and an aperiodic logarithmic signature for it, invoke the LogSig utility (Chapter 12) with the command line option "`-genaperiodic`" and the parameters "`-p:`" and "`-n:`".

For example, `LogSig.exe -genaperiodic -p:3 -n:10` can output $G = \mathbb{Z}_3 \oplus \mathbb{Z}_{27} \oplus \mathbb{Z}_9 \oplus \mathbb{Z}_3 \oplus \mathbb{Z}_{27}$ and $\alpha =$

$$\begin{pmatrix}(0,0,0,0,0)\\(0,0,3,0,0)\\(0,0,6,0,0)\\(1,0,0,0,0)\\(1,0,3,1,0)\\(1,0,6,2,0)\\(2,0,0,0,0)\\(2,0,3,1,0)\\(2,0,6,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0,0)\\(0,0,0,1,0)\\(0,0,0,2,0)\\(0,9,0,0,0)\\(0,9,6,1,0)\\(0,9,3,2,0)\\(0,18,0,0,0)\\(0,18,6,1,0)\\(0,18,3,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0,0)\\(2,3,6,1,9)\\(1,6,3,2,18)\\(2,12,0,2,0)\\(1,15,6,0,9)\\(0,18,3,1,18)\\(1,24,0,1,0)\\(0,0,6,2,9)\\(2,3,3,0,18)\end{pmatrix}\begin{pmatrix}(0,0,0,0,0)\\(0,8,8,2,15)\\(0,16,7,1,3)\\(2,6,8,2,0)\\(2,14,7,1,15)\\(2,22,6,0,3)\\(1,12,7,1,0)\\(1,20,6,0,15)\\(1,1,5,2,3)\end{pmatrix}\begin{pmatrix}(0,0,0,0,0)\\(1,24,6,0,8)\\(2,21,3,0,16)\\(0,8,4,1,9)\\(1,5,1,1,17)\\(2,2,7,1,25)\\(0,16,8,2,18)\\(1,13,5,2,26)\\(2,10,2,2,7)\end{pmatrix}.$$

Tameness. In Section 8.7, we will see that all logarithmic signatures generated by Algorithm 6.4 are tame.

6.4.2. Decomposition and Reunion

The following three algorithms (6.5, 6.6 and 6.7) have been presented in [Bau12].

Algorithm 6.5. [Bau12] Let G be an abelian group, $U \leq G$, R a transversal of U in G, $\delta := (D_1, \ldots, D_s)$ with $D_i := \{d_{i,1}, \ldots, d_{i,r_i}\} \subseteq R$ (i.e. $t(\delta) = (r_1, \ldots, r_s)$), $D_1 + \ldots + D_s = R$ and $|D_1| \cdots |D_s| = |R|$. Furthermore, we need logarithmic signatures

$$\alpha^{(j_1,\ldots,j_s)} := \left(A_1^{(j_1)}, \ldots, A_s^{(j_s)}\right) \in \Lambda(U)$$

for all $(j_1, \ldots, j_s) \in \{1, \ldots, r_1\} \times \ldots \times \{1, \ldots, r_s\}$.

Then $\beta := (B_1, \ldots, B_s)$ with

$$\begin{aligned} B_1 &:= (d_{1,1} + A_1^{(1)}) \cup \ldots \cup (d_{1,r_1} + A_1^{(r_1)}), \\ &\vdots \\ B_s &:= (d_{s,1} + A_s^{(1)}) \cup \ldots \cup (d_{s,r_s} + A_s^{(r_s)}) \end{aligned}$$

is a logarithmic signature for G.

In [Bau12], conditions for the blocks $A_i^{(j)}$ are given, such that β is aperiodic.

Note that the algorithm also works for non-abelian groups where $U \subseteq Z(G)$ or $R \subseteq Z(G)$.

Algorithm 6.6. [Bau12] Let $G = \mathbb{Z}_2^n$. Decompose G as

$$G = \underbrace{U_1 \oplus \ldots \oplus U_s}_{U} \oplus \underbrace{D_1 \oplus \ldots \oplus D_s}_{R},$$

where $U_1 \oplus D_1$ is a small group for which an aperiodic logarithmic signature β' is known. Furthermore, assume that $2 \leq |D_i| < \prod_{j=1}^{i-1} |U_j|$ for $i \in \{2, \ldots, s\}$.

Let $r_i := |D_i|$. For every $i \in \{2, \ldots, s\}$, choose a subset $K_i := \{k_i^{(1)}, \ldots, k_i^{(r_i)}\} \subseteq (U_1 \oplus \ldots \oplus U_{i-1}) \setminus \{(0, \ldots, 0)\}$.

In order to obtain an aperiodic logarithmic signature $\beta := (\beta', B_2, \ldots, B_s) \in \Lambda(G)$, use Algorithm 6.5 with $\delta := (D_2, \ldots, D_s)$ and

$$A_i^{(j)} := \{(0, \ldots, 0)\} \cup \left\{k_i^{(j)} + u \mid u \in U_i \setminus \{(0, \ldots, 0)\}\right\}$$

for $i \in \{2, \ldots, s\}$ and $j \in \{1, \ldots, r_i\}$.

Algorithm 6.7. [Bau12] Let $G = \mathbb{Z}_2^n$. Use Algorithm 6.6 to generate an aperiodic logarithmic signature $\beta \in \Lambda(G)$. Subsequently, apply a finite number of the transformations from Section 6.3 (i.e. element shuffles, block shuffles, block translations, block fusions/amalgamations) onto β.

The idea of applying these transformations is to hide the blocks of β' (which generate $U_1 \oplus D_1$), because finding these blocks results in the logarithmic signature being tame.

Tameness. As we will show in Theorem 8.39, all logarithmic signatures generated by Algorithm 6.7 are tame (because the transformations are insufficient to hide β').

6.4.3. Strongly Aperiodic Logarithmic Signatures

The following algorithm for generating strongly aperiodic logarithmic signatures has been proposed in [Sta13].

Algorithm 6.8. [Sta13] Let $G = \mathbb{Z}_p^{3s}$ with $p \in \mathbb{P}$, $s \geq 2$. Let $v_1, \ldots, v_{3s}$ be a generator set for G.

Apply Algorithm 6.5 with

$$\begin{aligned}
U &:= \langle v_1, \ldots, v_{2s} \rangle, \\
D_i &:= \{(0, \ldots, 0), v_{2s+i}, 2v_{2s+i}, \ldots, (p-1)v_{2s+i}\} \text{ for } i \in \{1, \ldots, s\}, \\
A_1^{(1)} &:= \langle v_1, v_2 \rangle, \\
A_1^{(j)} &:= \left\langle v_1 + v_2 + (j-1) \cdot \sum_{l=2}^{s} v_{2l},\ u \cdot v_2 + (j-1) \cdot \sum_{l=2}^{s} v_{2l-1} \right\rangle \text{ for } j \in \{2, \ldots, p\}, \\
A_i^{(j)} &:= \langle v_{2i-1} + (j-1)v_1,\ v_{2i} + (j-1)v_2 \rangle \text{ for } i \in \{2, \ldots, s\}, j \in \{1, \ldots, p\},
\end{aligned}$$

where $u \in \mathbb{Z}_p \setminus \{0\}$ is chosen such that the polynomial $x^2 - x - u \in \mathbb{Z}_p[x]$ has no root in $\mathbb{Z}_p$.

The resulting logarithmic signature β (of type $t(\beta) = (p^3, \ldots, p^3)$) is aperiodic, and furthermore fusing any $s-1$ or less blocks of β results in another aperiodic logarithmic signature (i.e. β is strongly aperiodic).

In [Sta13], the authors moreover present a similar algorithm generating strongly aperiodic logarithmic signatures of type $(2^3, 2^2, \ldots, 2^2)$ for $\mathbb{Z}_2^{2s-1}$.

Tameness. In Section 8.9.3 we show that all logarithmic signatures generated by Algorithm 6.8 (and the ones for $\mathbb{Z}_2^{2s-1}$) are tame.

6.5. LS-Gen

We now propose and analyze a new algorithm called *LS-Gen*. LS-Gen works for both abelian and non-abelian groups. For abelian groups, LS-Gen typically generates more logarithmic signatures than the algorithm for generating amalgamated logarithmic signatures (Section 6.3).

Our algorithm starts with a fixed tame logarithmic signature α of a group G and generates another logarithmic signature β. Without knowing how β was generated, β might be wild (depending on various things like the group, block sizes of α, etc.).

In order to generate β, various transformations are applied to the logarithmic signature. In Section 6.5.3 we describe these transformations in detail (their input, output, properties, etc.). From the transformations that are applicable for G, LS-Gen randomly selects one and applies it to the current logarithmic signature. This is repeated a few times (called *rounds*).

Not all of the transformations might be applicable to a given group. An example is the transformation permuting the order of a range of blocks in the logarithmic signature. This operation might always be possible (e.g. when G is abelian) or sometimes only (e.g. when $G = D_{2n}$). Deciding whether blocks can be permuted freely can be hard in general (without knowing anything about the structure of the group). However, the implementation of the group can often easily decide this.

Therefore, we leave such tests/operations up to the implementation of the group, and call them *group implementation capabilities*. Such capabilities are described in Section 6.5.1.

LS-Gen is designed to be extensible. Transformations, input/output relations and group implementation capabilities can be added easily. When adding new transformations, security usually can only increase, not decrease (the previous transformations can still be selected, i.e. the statements in Section 6.5.5 are still valid).

6.5.1. Group Implementation Capabilities

Group implementation capabilities are tests/operations that the implementation of a group supports. A group implementation is not required to support all of the capabilities listed below, but the more capabilities it supports, the more transformations LS-Gen can use (and thus possibly generate stronger logarithmic signatures).

The ability to multiply elements is not considered to be a group implementation capability. All implementations must support this operation.

With this and G being finite, there also automatically is an efficient algorithm available for inverting elements: the inverse of an arbitrary $g \in G$ is the element $g^{|G|-1}$, which can be computed efficiently in time polynomial in $\log_2 |G|$ by a square-and-multiply (binary exponentiation) algorithm. If the group implementation does not provide a custom method for inverting elements, such an algorithm can be used. Of course, usually a group implementation will override this and provide an even more efficient element inversion method.

In object-oriented programming languages, group implementation capabilities can be realized using interfaces. A group would be realized as a class, and it supports a group implementation capability if it implements the corresponding interface. This is how we have implemented it in our LogSig utility (see Chapter 12); all the group implementation capability interfaces can be found in the namespace `LogSig.DM.LsGen.GroupImplCaps`.

In non-object-oriented programming languages, group implementation capabilities can be realized using a system based on function queries, e.g. similar to the OpenGL extension system (using `wglGetProcAddress` or `glXGetProcAddress` to obtain extension function pointers).

In the list of group implementation capabilities below, "Interface in LogSig" speci-

fies the name of the interface by which we have realized this capability definition in LogSig.

Note that a "block sequence" is not necessarily a part of the current logarithmic signature, but may be any sequence of blocks containing group elements.

Group implementation capabilities:

- *Name:* `FacPermShuffleTest`.
  Input: block sequence.
  Output: boolean value.
  Interface in LogSig: `IFacPermShuffleTest`.
  Description: Test whether every block shuffle of the input block sequence is factorization-permuting (see Section 5.1.2). This test is allowed to be one-sided: when it returns *true*, every block shuffle is guaranteed to be factorization-permuting, however when it returns *false*, a block shuffle may or may not be factorization-permuting.

  Examples: If the group is abelian, the test can simply return *true* (without even looking at the input blocks). In the case of a dihedral group, the test could check whether the blocks contain rotations only (this would be a one-sided test as there could be more blocks that are factorization-permuting when being shuffled, e.g. $\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\tau}$ in $D_{2\cdot 8}$).

- *Name:* `NormalChainTest`.
  Input: block sequence.
  Output: boolean value.
  Interface in LogSig*:* `INormalChainTest`.
  Description: Test whether the input block sequence $A_1, \ldots, A_k$ fulfills all of the following properties: $A_1 \trianglelefteq G, A_1A_2 \trianglelefteq G, \ldots, A_1 \cdots A_k \trianglelefteq G$. This test is allowed to be one-sided: when it returns *true*, the block sequence fulfills all of the above properties, however when it returns *false*, the block sequence may or may not fulfill the properties.

- *Name:* `NormalChainFac`.
  Input: block sequence, group element g.
  Output: integer array.
  Interface in LogSig*:* `INormalChainFac`.
  Description: If `NormalChainTest` indicates that a block sequence has the specified structure, then the `NormalChainFac` capability allows factoring elements in the block sequence: it returns the factorization indices of g with respect to the input block sequence.

6.5.2. Transformation Input/Output Relations

Transformation input/output relations specify the connection between element factorizations with respect to the input and output logarithmic signatures of a transformation.

Usually a transformation is able to update element factorizations directly: when a factorization of an element $g \in G$ is passed together with the respective logarithmic signature to the transformation, the transformation on the one hand outputs the transformed logarithmic signature and on the other hand the updated factorization of g, i.e. an array of indices such that the product of the element at these positions in the new logarithmic signatures again is g. However, this is not possible with all transformations. For example, consider a left translation of a logarithmic signature, then the factorization of a $g \in G$ cannot be derived easily from the input logarithmic signature. This is where transformation input/output relations come into play. In the left translation example, the relation would be simple: in order to factor an element $g \in G$ in the new logarithmic signature, we can factor $t^{-1}g$ (where t is the left translation element) in the input logarithmic signature and select the same indices in the new logarithmic signature to obtain the factorization of g.

When LS-Gen wants to factor an element $g \in G$, it can undo all the transformation input/output relations from the end up to the start in order to compute the element that needs to be factored in the starting, tame logarithmic signature (such that after the generation process we end up with the factorization of g in the final logarithmic signature).

In our LogSig utility, all transformation input/output relations can be found in the namespace `LogSig.DM.LsGen.TransformationIORels`.

Transformation input/output relations:

- *Name*: `LeftRightTranslationRel`.
  Impl. in LogSig*:* `LeftRightTranslationRel`.
  Description: The input/output elements are related by left and right translations. The object representing this relation needs to save two group elements l and r that represent the left/right translations. For an input element $g \in G$, the transformation outputs the factorization of lgr. Conversely, if we want the factorization of an element $h \in G$ in the output logarithmic signature, we need to factor $l^{-1}hr^{-1}$ in the input logarithmic signature.

Merging. Multiple consecutive transformation input/output relations (especially ones of the same type) may be merged to a single one. Merging transformation input/output relations can save space.

LogSig merges consecutive `LeftRightTranslationRel`s to a single one (by multiplying the translation elements).

6.5.3. Transformations

Every transformation gets a structure containing the following objects as parameter:

- **Group implementation object.** Using this object, group elements may be multiplied, inverted, etc.

- **CSPRNG instance.** An instance of a cryptographically secure pseudo-random number generator that the transformation should use when random numbers are required.

- **Logarithmic signature.** The current logarithmic signature. A transformation may modify this object. On input and output, the logarithmic signature must be canonical (i.e. id is the first element in every block).

- **List of transformation input/output relations.** If the current transformation requires a relation, it is supposed to add the relation to the end of this list.

- **Factorization (array of indices).** This array is optional from a caller point of view. If no array is specified, the transformation can ignore this parameter. If an array is specified (corresponding to the factorization of an element g in the input logarithmic signature), the transformation must update the indices, i.e. after the transformation multiplying the elements at these indices in the output logarithmic signature must again result in g or in an element related to g by a transformation input/output relation that the transformation added to the list of relations.

In the LogSig utility, an implementation of this structure is the class `TransformationParams` in the `LogSig.DM.LsGen` namespace.

Saving the list of transformation input/output relations is optional (the list can be regenerated when knowing the CSPRNG seed), but searching for factorizations of elements is faster when the list is known at the start. If the list is present (for all transformations), the relations can be undone from the end to the start in order to obtain the element that we need to start with, and then apply all the transformations in order to end up with a factorization of the element that we originally were looking for. If the relations list is not present, LS-Gen needs to be run once in order to generate the relations list and then again to compute the factorization like above, i.e. two runs of LS-Gen are required in this case.

In the following, we list all transformations supported by LS-Gen. Some transformations require group implementation capabilities; if the group implementation does not support the required capability, the transformation will not be used.

In the list below, "Impl. in LogSig" specifies the name of the class by which the transformation is implemented in our LogSig utility. All transformations can be found in the namespace `LogSig.DM.LsGen.Transformations`.

If no group implementation capabilities are specified, then the transformation is always applicable (i.e. for every group). Only the group implementation capabilities in Section 6.5.1 are listed; standard operations like multiplying/inverting elements do not count.

If no transformation input/output relations are specified, then the transformation leaves the relations list in the parameters unmodified and updates the factorization indices array (if passed by the caller) appropriately, such that the factorization subsequently specifies the same element as on input.

Transformations:

- **Block substitution.**
  Impl. in LogSig*:* `LsgtBlockSubst`.
  Description: Performs block substitutions. See Section 5.1.8 for properties and implementation details. The transformation performs n block substitutions at randomly generated positions (where n is number of blocks of α).

- **TSN transformation.**
  Trf. I/O rel.: `LeftRightTranslationRel`.
  Impl. in LogSig*:* `LsgtTsn`.
  Description: Performs a TSN transformation as specified in Section 5.1.13, either from the left to the right or from the right to the left (based on a bit drawn from the CSPRNG). The I/O relation object stores the left/right translation and the inverse of the element in the last/first block that is removed at the very end by the normalization (if the last/first block contains an element h, removing the last/first block is equivalent to right-/left-translating the logarithmic signature by h^{-1}).

- **Block shuffle.**
  Req. group impl. caps.: `FacPermShuffleTest`.
  Impl. in LogSig*:* `LsgtBlockShuffle`.
  Description: Select a random block. Then iteratively try to extend the selection to the left and to the right as long as the `FacPermShuffleTest` returns *true*, i.e. as long as the selected blocks are factorization-permuting. When the selection cannot be extended anymore (i.e. when reaching the leftmost/rightmost block or the `FacPermShuffleTest` returning *false* when trying to include one more block), permute the selected blocks randomly. If a factorization array is specified by the caller, permute the factorization indices accordingly (using the same permutation that was used to permute the blocks of the logarithmic signature); this is possible, because the block sequence being permuted is guaranteed to be factorization-permuting.

- **Normal chain selective shifts.**
  Req. group impl. caps.: `NormalChainTest`, `NormalChainFac`.
  Impl. in LogSig*:* `LsgtNormalSelShifts`.
  Description: This transformation first searches for a chain of normal subgroups and then performs selective shifts (see Section 5.1.9), i.e. it multiplies chain elements onto elements in other blocks.

  In detail: first generate a random permutation $\pi \in \mathrm{Sym}(n)$ (where n is the number of blocks in the current logarithmic signature). While iterating i from 1 to n, test whether the block $A_{\pi(i)}$ is a normal subgroup using the `NormalChainTest` group

implementation capability. If it is a normal subgroup, add $\pi(i)$ to a list L (which initially is empty). Then iterate i from 1 to n and for all $i \notin L$ test whether the block sequence $(A_{L[0]}, A_{\pi(i)})$ is a normal subgroup chain (using `NormalChainTest` again). If it is, add $\pi(i)$ to L. Continue like this until `NormalChainTest` returns *false* for all remaining blocks. Now, generate a random number $0 \leq d < |L|$ and remove the last d blocks from L. Multiply randomly selected elements from the remaining chain L onto elements in other blocks (in order to select an element randomly in the chain, generate a random index for each chain block and multiply the elements at these indices).

If the caller has specified an element factorization array and a chain element is multiplied onto an element being part of the factorization, then update the chain factorization elements using `NormalChainFac`: when the factorization indices in the chain blocks currently result in a product g and a chain element h is multiplied onto an element outside the chain blocks that is part of the factorization, let `NormalChainFac` compute the factorization indices of gh^{-1} and copy these into the element factorization array.

6.5.4. Algorithm

Let $r \in \mathbb{N}$ a security level parameter. r specifies how many transformations are applied, similar to the number of rounds in an iterated block cipher. Depending on G, all logarithmic signatures might be tame, and thus the algorithm of course also only can create tame ones (i.e. increasing the transformations number r does not increase security).

LS-Gen requires random numbers, thus a CSPRNG s must be supplied. The seed of the CSPRNG corresponds to the private key of the system. Knowing the seed, the way how the final logarithmic signature was generated is fully defined.

The group element g and transformation input/output relations list l in the parameters are optional. However, when a g is specified, the algorithm also expects l to be present and valid. Thus, if the caller does not know l, he first has to run LS-Gen once without a g in order to obtain l and subsequently run it again with g and l in the parameters.

Function `LS-Gen`(Group G, UInt r, CSPRNG s, GroupElement g, List<TrfIORel> l) : LogSig × List<UInt>

1. Prepare the transformations parameter structure S as described in Section 6.5.3, using the function parameters of `LS-Gen`.

   The logarithmic signature in S is set to a fixed, tame logarithmic signature of G (provided by the implementation of G), in which factorizations can be computed easily (also done by the implementation of G).

   If $g \neq null$:

   - Undo the transformation input/output relations in l from the end to the beginning (starting at the end with g) in order to obtain an element g'. Set the

factorization indices array in S to the factorization of g' in the initial tame logarithmic signature.

2. Repeat r times:
   - Randomly choose one of the applicable transformations (i.e. the ones for which all required group implementation capabilities are available) and execute it on S.
3. Copy the transformation input/output relations from the list in S to l.
4. If $g \neq null$: `return` (S.LogSig, S.Factorization),
   else: `return` (S.LogSig, *null*).

6.5.5. Analysis

Round count r. Depending on G, a rather small r might suffice or a large r might be required to possibly achieve reasonable security.

CSPRNG s. The PRNG s should be cryptographically secure.

- s should have a large internal state/seed. If it has a small internal state/seed only and r is known or small, LS-Gen can produce only a small number of different logarithmic signatures (polynomial in the input length). In such a case all generated logarithmic signatures are tame, because an attacker can regenerate them all.
- LS-Gen is extensible and should be secure even when more transformations are added. Such transformations may leak information on the random numbers generated by s, which could be a problem, if s would not be cryptographically secure.

  For example, consider a transformation that generates 1 random bit for each block of the current logarithmic signature and sorts the elements within each block in ascending or descending order, depending on the random bit. If the current logarithmic signature has n blocks and this transformation is applied at the very end, n consecutive output bits of s are leaked. If an attacker can use these bits to strongly limit the number of possible seeds or even reconstruct the seed of s (e.g. for LFSRs of length n with fixed coefficients this is possible easily), the logarithmic signature is tame, because the attacker can then regenerate the logarithmic signature using LS-Gen.

Number of adjacent blocks. In the block substitution transformation it might suffice to always replace only two adjacent blocks by equivalent ones. For example with type $(2, 2, \ldots, 2)$ signatures of $D_{2 \cdot 2^n}$ this is the case, but more blocks might be required for other groups.

Transformations are not expressible by each other. Block substitutions and TSN transformations do not require any group implementation capabilities, i.e. they are

always applicable (independent of the group), thus especially for these two their interplay is interesting.

A block substitution cannot be expressed by a series of TSN transformations. Conversely, a TSN transformation cannot be expressed by a series of block substitutions.

We show this with two simple examples.

Let $G = \mathbb{Z}_2^2$ and $\alpha = \binom{(0,0)}{(1,0)}\binom{(0,0)}{(0,1)} \in \Lambda(G)$. One possible block substitution is swapping the two blocks: $\alpha' := \binom{(0,0)}{(0,1)}\binom{(0,0)}{(1,0)}$ is indeed a logarithmic signature for G, too. However, α' cannot be generated from α using a TSN transformation, because α is invariant under TSN transformations by Corollary 5.17.

Let $G = \mathbb{Z}_8$ and $\alpha = \binom{0}{1}\binom{0}{2}\binom{0}{4}$. With a TSN transformation, we can achieve that one block contains the element 7: skip the left translation, swap the two elements within the first block, normalize and discard the right translation; we obtain: $\alpha' := \binom{0}{7}\binom{0}{2}\binom{0}{4}$. It is impossible to generate α' using block transformations where only two adjacent blocks are substituted (obviously if we would allow three blocks being substituted, α' could be generated). In order to see this, let us have a look at the possible changes that a block substitution can do. First of all, it can freely permute blocks (G is abelian) and elements within blocks. It can add the 4 to 1 and 2, because $\{0, 4\}$ is a subgroup of G. 1 and 2 cannot be added to any other value, because this would violate the structural observation for $\mathbb{Z}_{2^n}$ in Lemma 8.4. So, the only elements that we will see after block substitutions are $0, 1, 2, 4, 5$ and 6. 3 and 7 cannot be generated, however α' contains a 7, thus a TSN transformation cannot be expressed using block substitutions.

A larger set than $\mathcal{AT}(G)$ is generated for most abelian groups. Let G be an abelian group. LS-Gen typically generates a proper superset of $\mathcal{AT}(G)$.

First of all, observe that all transformations in Section 6.3 can be expressed using the transformations of LS-Gen. Permuting blocks and elements within blocks can be realized using block substitutions, a translation can be realized using a TSN transformation, and blocks can be amalgamated using block substitutions. Thus, LS-Gen can generate all logarithmic signatures in $\mathcal{AT}(G)$.

To see why typically a proper superset is generated, we give an example of a logarithmic signature that LS-Gen can generate, but is not an element of $\mathcal{AT}(G)$. Let $G = \mathbb{Z}_3^4$ and $\alpha = (A_1, A_2)$ with

$$A_1 := \begin{pmatrix} (0,0,0,0) \\ (0,0,1,1) \\ (0,0,2,2) \\ (1,0,0,0) \\ (1,0,1,0) \\ (1,0,2,0) \\ (2,0,0,0) \\ (2,0,1,0) \\ (2,0,2,0) \end{pmatrix}, \quad A_2 := \begin{pmatrix} (0,0,0,0) \\ (0,0,2,1) \\ (0,0,1,2) \\ (0,1,0,0) \\ (0,1,0,1) \\ (0,1,0,2) \\ (0,2,0,0) \\ (0,2,0,1) \\ (0,2,0,2) \end{pmatrix}.$$

α is an aperiodic logarithmic signature for G. Obviously, α can be generated by LS-Gen (start with an arbitrary logarithmic signature of type $(3, 3, 3, 3)$ and apply block substitutions: amalgamate the left two blocks, amalgamate the right two blocks, and replace the remaining two blocks by the blocks above), but $\alpha \notin \mathcal{AT}(G)$, because all logarithmic signatures in $\mathcal{AT}(G)$ have a periodic block (as proven in Section 8.9.1).

7. Factoring in General

In Chapter 7, we have a look at factorization approaches for arbitrary groups, in a rather abstract way.

Our contributions. We show that regarding only canonical block sequences is not a real restriction. We consider using homomorphisms, both for logarithmic signatures and multiple factorizations. We furthermore consider moving into factor groups via normal subgroup blocks. A demonstration with direct products shows that a simple concatenation approach does not work.

7.1. Canonical Block Sequences

For many factorization algorithms we assume that the input block sequence is canonical. This is not really a restriction, as the following proposition shows.

Proposition 7.1. *Let G be a group and $\alpha \in \Xi_s(G)$. Let $\beta \in \Xi_s(G)$ be the block sequence obtained by normalizing α, extracting the right translation and discarding it (like in Section 5.1.13). Note that β is canonical.*

If β is tame, then α is tame, too.

Proof. Let $g \in G$ be an element to be factored with respect to α, and $\beta' \in \Xi_{s+1}(G)$ the block sequence obtained by normalizing α and extracting the right translation (to a new block of size 1). Then $\beta = (\beta'[1], \beta'[2], \ldots, \beta'[s])$. Let $r := \beta'[s+1][1]$.

Compute the factorization index vector $(i_1, \ldots, i_s) \in \mathbb{N}^s$ of $g \cdot r^{-1}$ with respect to β (which is possible efficiently, due to β being tame), such that $\beta[1][i_1] \cdot \beta[2][i_2] \cdots \beta[s][i_s] = g \cdot r^{-1}$. Then $(i_1, \ldots, i_s)$ also is a factorization index vector of g with respect to α:

$$\begin{aligned}
&\beta[1][i_1] \cdot \beta[2][i_2] \cdots \beta[s][i_s] = g \cdot r^{-1} \\
\Leftrightarrow\ &\beta[1][i_1] \cdot \beta[2][i_2] \cdots \beta[s][i_s] \cdot r = g \cdot r^{-1} \cdot r \\
\Leftrightarrow\ &\beta'[1][i_1] \cdot \beta'[2][i_2] \cdots \beta'[s][i_s] \cdot \beta'[s+1][1] = g \\
\overset{*}{\Leftrightarrow}\ &\alpha[1][i_1] \cdot \alpha[2][i_2] \cdots \alpha[s][i_s] = g.
\end{aligned}$$

(*) A normalization is a sandwich, and sandwiching does not change factorization index vectors. □

Example 7.2. Let $G = D_{2 \cdot 3^3}$, $\alpha = \begin{pmatrix} 7\tau \\ 15 \\ 13 \end{pmatrix} \begin{pmatrix} 4\tau \\ 22\tau \\ 10 \end{pmatrix} \begin{pmatrix} 2 \\ 1\tau \end{pmatrix} \begin{pmatrix} 2 \\ 14 \\ 9\tau \end{pmatrix} \in \Lambda(G)$ and $g = 11\tau$.

First, we normalize α, extract the right translation and discard it:

$$\begin{aligned}
\alpha &= \begin{pmatrix}7\tau\\15\\13\end{pmatrix}\begin{pmatrix}4\tau\\22\tau\\10\end{pmatrix}\begin{pmatrix}2\\1\tau\end{pmatrix}\begin{pmatrix}2\\14\\9\tau\end{pmatrix}\\
&\rightsquigarrow \begin{pmatrix}\mathrm{id}\\22\tau\\20\tau\end{pmatrix}\begin{pmatrix}3\\12\\24\tau\end{pmatrix}\begin{pmatrix}2\\1\tau\end{pmatrix}\begin{pmatrix}2\\14\\9\tau\end{pmatrix}\\
&\rightsquigarrow \begin{pmatrix}\mathrm{id}\\22\tau\\20\tau\end{pmatrix}\begin{pmatrix}\mathrm{id}\\9\\\tau\end{pmatrix}\begin{pmatrix}5\\4\tau\end{pmatrix}\begin{pmatrix}2\\14\\9\tau\end{pmatrix}\\
&\rightsquigarrow \begin{pmatrix}\mathrm{id}\\22\tau\\20\tau\end{pmatrix}\begin{pmatrix}\mathrm{id}\\9\\\tau\end{pmatrix}\begin{pmatrix}\mathrm{id}\\9\tau\end{pmatrix}\begin{pmatrix}7\\19\\14\tau\end{pmatrix}\\
&\rightsquigarrow \begin{pmatrix}\mathrm{id}\\22\tau\\20\tau\end{pmatrix}\begin{pmatrix}\mathrm{id}\\9\\\tau\end{pmatrix}\begin{pmatrix}\mathrm{id}\\9\tau\end{pmatrix}\begin{pmatrix}\mathrm{id}\\12\\21\tau\end{pmatrix}(7) = \beta',\\
\beta &= \begin{pmatrix}\mathrm{id}\\22\tau\\20\tau\end{pmatrix}\begin{pmatrix}\mathrm{id}\\9\\\tau\end{pmatrix}\begin{pmatrix}\mathrm{id}\\9\tau\end{pmatrix}\begin{pmatrix}\mathrm{id}\\12\\21\tau\end{pmatrix},\\
r &= 7.
\end{aligned}$$

We have $g \cdot r^{-1} = 11\tau \cdot (-7) = 11\tau \cdot 20 = 18\tau$. The factorization of 18τ with respect to β is

$$\beta = \begin{pmatrix}\boxed{\mathrm{id}}\\22\tau\\20\tau\end{pmatrix}\begin{pmatrix}\mathrm{id}\\\boxed{9}\\\tau\end{pmatrix}\begin{pmatrix}\mathrm{id}\\\boxed{9\tau}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\12\\21\tau\end{pmatrix}.$$

By selecting the elements at the same indices in α, we obtain:

$$\alpha = \begin{pmatrix}\boxed{7\tau}\\15\\13\end{pmatrix}\begin{pmatrix}4\tau\\\boxed{22\tau}\\10\end{pmatrix}\begin{pmatrix}2\\\boxed{1\tau}\end{pmatrix}\begin{pmatrix}\boxed{2}\\14\\9\tau\end{pmatrix}.$$

We get $7\tau \cdot 22\tau \cdot 1\tau \cdot 2 = 11\tau = g$, so this indeed is the factorization of g with respect to α.

7.2. Homomorphisms and Multiple Factorizations

Theorem 7.3. *Let G, H be groups, $\varphi : G \to H$ a surjective group homomorphism, $k := |\ker(\varphi)|$, and $\alpha = (A_1, \ldots, A_s)$ an l-factorization of G. Then $\varphi(\alpha)$ is a $(k \cdot l)$-factorization of H (i.e. every element of H can be expressed in exactly $k \cdot l$ different ways with respect to the block sequence $\varphi(\alpha)$).*

Proof. As φ is a surjective group homomorphism, we have $G/\ker(\varphi) \cong H$ by the isomorphism theorem. Thus every $h \in H$ has exactly k preimages under φ: let h' be any preimage of h under φ (i.e. $\varphi(h') = h$), then the set of all preimages of h under φ is $\varphi^{-1}(h) = \{z \cdot h' \mid z \in \ker(\varphi)\}$.

Let $(i_1, \ldots, i_s)$ be a factorization index vector of an element $g \in G$, i.e. $g = A_1[i_1] \cdots A_s[i_s]$. Then $\varphi(A_1[i_1]) \cdots \varphi(A_s[i_s])$ is a factorization of $\varphi(g)$ in $\varphi(\alpha)$, because $\varphi(A_1[i_1]) \cdots \varphi(A_s[i_s]) = \varphi(A_1[i_1] \cdots A_s[i_s]) = \varphi(g)$. As α is an l-factorization of G, all of the k elements in $\varphi^{-1}(h)$ have l different factorizations in α; these correspond to the $k \cdot l$ different factorizations of h with respect to $\varphi(\alpha)$. So, $\varphi(\alpha)$ is a $(k \cdot l)$-factorization of H. □

Theorem 7.4. *Let G, H be groups, $\varphi : G \to H$ a surjective group homomorphism, $k := |\ker(\varphi)|$, and α an l-factorization of G. If k and l are polynomial in $\ell(\alpha)$ and if we can efficiently factor elements in the $(k \cdot l)$-factorization $\varphi(\alpha)$ of H, then we can efficiently factor elements with respect to α.*

Proof. Suppose we want to factor a $g \in G$. Apply φ to the elements in the blocks of α to obtain $\varphi(\alpha)$. Compute the $k \cdot l$ different factorizations of the element $\varphi(g)$ in $\varphi(\alpha)$. Selecting the same indices in α, these $k \cdot l$ different factorizations generate the elements $\{z \cdot g \mid z \in \ker(\varphi)\}$ (each element of this set is generated exactly l times). As $\mathrm{id} \in \ker(\varphi)$, l of these $k \cdot l$ different factorizations generate g. □

Remark 7.5. Applying Theorem 7.4 recursively does not result in an efficient factorization algorithm (even when the homomorphism kernels on each recursion level are of fixed size), because l grows exponentially.

For example, let $G = \mathbb{Z}_{2^n}$ and $\alpha \in \Lambda(G)$ canonical of type $(2, 2, \ldots, 2)$. By Rédei's theorem (Theorem 8.1), α contains a subgroup block A. The canonical homomorphism $\varphi : G \to G/A : g \mapsto g + A$ has a kernel of size 2, $G/A \cong \mathbb{Z}_{2^{n-1}}$ and $\varphi(\alpha)$ is a 2-factorization of G/A. Repeating this, we obtain a 4-factorization of $\mathbb{Z}_{2^{n-2}}$ (by Theorem 8.29 there must be a subgroup block in the multiple factorization), then an 8-factorization of $\mathbb{Z}_{2^{n-3}}$, and so on. We end up with an 2^{n-1}-factorization of $\mathbb{Z}_2$. As this is exponential in n, computing these 2^{n-1} factorizations and passing them up to the recursion level above cannot be done efficiently.

7.3. Homomorphisms and Normal Subgroup Blocks

Theorem 7.6. *Let G be a group and $\alpha = (A_1, \ldots, A_s) \in \Lambda(G)$ with at least one block A_i being a normal subgroup in G. Define α' by removing block A_i from α and replacing all elements g by $g \cdot A_i$ in the remaining blocks, then $\alpha' \in \Lambda(G/A_i)$.*

If factorizations in α' can be computed efficiently, then factorizations in α can be computed efficiently, too.

Proof. Write $N := A_i$ and let $\varphi : G \to G/N : g \mapsto gN$ be the canonical group homomorphism (projection) of G onto G/N. By Theorem 7.3, the block sequence $\varphi(\alpha)$ is an

$|N|$-factorization of G/N (i.e. every element in G/N is expressible by exactly $|N|$ different factorizations). α' is the block sequence obtained from $\varphi(\alpha)$ by removing the ith block. Observe that $E(\varphi(A_i)) = \{\mathrm{id}\}$ (as all elements are in N), so α' is a logarithmic signature for G/N.

In order to factor an element $x \in G$, first factor xN in α' (by hypothesis we can factorize in α' in polynomial time). Select the corresponding elements in α (the elements at the same positions as in α' except A_i). Now, compute the product (in G) of the elements selected in the blocks $A_1, \ldots, A_{i-1}$ and call it y_l. Analogously, compute the product (in G) of the elements selected in the blocks $A_{i+1}, \ldots, A_s$ and call it y_r. There is exactly one element $a \in A_i$, such that $y_l \cdot a \cdot y_r = x$, which completes the factorization of x that we were looking for. □

Remark 7.7. Especially for non-abelian groups, this approach usually does not directly lead to a factorization algorithm, because encountering a normal subgroup block in a logarithmic signature is a rather rare event. However, it can be part of a factorization algorithm: if the logarithmic signature can be transformed (in an invertible way) such that a block becomes a normal subgroup, the above can be used as simplification, and possibly this leads to an efficient factorization algorithm.

Note that for this it is also required to be able to efficiently compute in the factor group. It is not immediately clear whether or how this is possible for an arbitrary group.

Theorem 7.6 can be generalized using homomorphisms:

Theorem 7.8. *Let G, H be groups, $\alpha = (A_1, \ldots, A_s) \in \Lambda(G)$, $\varphi : G \to H$ a surjective group homomorphism with $k := |\ker(\varphi)|$ polynomial in $\ell(\alpha)$, and $I \subseteq \{1, \ldots, s\}$ a set of indices such that $\prod_{i \in I} |A_i| = k$ and $\bigcup_{i \in I} A_i \subseteq \ker(\varphi)$ (in particular, $\prod_{i \in I} A_i = \ker(\varphi)$).*

Define β by removing all blocks A_i with $i \in I$ from α and replacing all elements g by $\varphi(g)$ in the remaining blocks, then $\beta \in \Lambda(H)$.

If factorizations with respect to β can be computed efficiently, then factorizations with respect to α can be computed efficiently, too.

Proof. Given an element $g \in G$, first factor $\varphi(g)$ with respect to β (which is possible efficiently by hypothesis). Since $E(\varphi(A_i)) = \{\mathrm{id}\}$ for all $i \in I$, in the factorization of g with respect to α the same indices need to be selected in the blocks A_j with $j \notin I$. In order to complete the factorization, elements in the blocks A_i with $i \in I$ need to be selected. There are k possibilities, and trying them using brute-force (i.e. for each possibility multiplying the selected elements and testing whether the product gives g) is possible efficiently, because k is polynomial in $\ell(\alpha)$ by hypothesis. □

7.4. Direct Products

Let H, K be groups and $G := H \times K$. Let $\varphi_H : G \to H : (h, k) \mapsto h$ and $\varphi_K : G \to K : (h, k) \mapsto k$. Let $\psi_H : H \to G : h \mapsto (h, \mathrm{id})$ and $\psi_K : K \to G : k \mapsto (\mathrm{id}, k)$.

Combining logarithmic signatures. Let $\beta \in \Lambda(H)$ and $\gamma \in \Lambda(K)$. Then

clearly $\alpha := \psi_H(\beta)\psi_K(\gamma) \in \Lambda(G)$.

Factoring. Let $\alpha \in \Lambda(G)$ such that the blocks can be partitioned into two sets, where the first set forms a logarithmic signature for H after applying φ_H and the second set forms a logarithmic signature for K after applying φ_K. In general it is insufficient to factor $\varphi_H(g)$ with respect to the logarithmic signature for H, respectively $\varphi_K(g)$ with respect to the logarithmic signature of K and combine the factorization indices.

For example, let $H = \mathbb{Z}_{2^2}, K = \mathbb{Z}_{3^2}$, $G = H \oplus K$,

$$\alpha = \underbrace{\begin{pmatrix}(0,0)\\(1,3)\end{pmatrix}\begin{pmatrix}(0,0)\\(2,0)\end{pmatrix}}_{\beta}\underbrace{\begin{pmatrix}(0,0)\\(2,1)\\(0,2)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,3)\\(0,6)\end{pmatrix}}_{\gamma} \in \Lambda(G).$$

This α was constructed by taking canonical exact transversal logarithmic signatures for H and K and performing selective shifts from the fourth block onto the first one and from the second block onto the third one.

Clearly, $\varphi_H(\alpha)$ is a 9-factorization of H; $\varphi_H(\beta)$ is a logarithmic signature for H. Analogously, $\varphi_K(\alpha)$ is a 4-factorization of K; $\varphi_K(\gamma)$ is a logarithmic signature for K.

Let $g = (1,1) \in G$. The factorization index vector for g with respect to α is $(2,2,2,3)$ (since $(1,3) + (2,0) + (2,1) + (0,6) = (1,1)$).

The factorization index vector of $1 \in H$ with respect to $\varphi_H(\beta)$ is $(2,1)$. The factorization index vector of $1 \in K$ with respect to $\varphi_K(\gamma)$ is $(2,1)$.

Observe that although $\varphi_H(\beta) \in \Lambda(H)$ and $\varphi_K(\gamma) \in \Lambda(K)$, the correct factorization index vector for g with respect to α is not obtained by concatenating the factorization index vectors for $\varphi_H(g)$ with respect to $\varphi_H(\beta)$ and $\varphi_K(g)$ with respect to $\varphi_K(\gamma)$: the correct vector is $(2,2,2,3)$ and concatenating would result in $(2,1,2,1)$.

8. Abelian Groups

In Chapter 8, we regard factorizations of abelian groups G, represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$.

Our contributions. Our contributions in this chapter start with a detailed analysis of the structure of logarithmic signatures $\alpha \in \Lambda(\mathbb{Z}_{2^n})$ of type $t(\alpha) = (2, 2, \ldots, 2)$. This leads to an efficient factorization algorithm for logarithmic signatures $\alpha \in \Lambda(\mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^k)$ of type $t(\alpha) = (2, 2, \ldots, 2)$.

Based on a theorem of Rédei, we show that for all abelian groups every logarithmic signature $\alpha = (A_1, A_2, \ldots, A_n) \in \Lambda(G)$ with $|A_i| \in \mathbb{P}$ for all $1 \leq i \leq n$ is tame. This especially includes a detailed elaboration on how to efficiently compute in factor groups.

Building upon the previous result, we design more factorization algorithms. First, we drop the requirement that blocks need to be of prime size and point out that the previous algorithm still works fine if it finds a subgroup on each recursion level. We further generalize this by showing that finding a periodic block on each recursion level is sufficient.

We then design a generic factorization algorithm, which is defined for all block sequences (e.g. the input block sequence does not necessarily have to contain a periodic block). We justify why this algorithm is efficient when periodic blocks exist on each recursion level (i.e. the generic algorithm supersedes all previous factorization algorithms), argue why it is efficient even for aperiodic logarithmic signatures generated by an algorithm in Chapter 6, and point out why it even may be efficient for logarithmic signatures generated by LS-Gen in practice.

We have a look at other factorization approaches, e.g. via recursively moving into statically or dynamically chosen factor groups, and demonstrate obstacles that can occur.

Using our algorithms, we show that various classes of logarithmic signatures are tame now: amalgamated transversal, aperiodically decomposed and reunited for $\mathbb{Z}_2^n$ from [Bau12], and strongly aperiodic constructions from [Sta13].

We provide a list of logarithmic signatures (of specific types and for specific groups) that can be proved to be tame.

We describe how a factorization problem can be modeled as an integer linear programming (ILP) problem. It turns out that none of the ILP solvers that we tested are able to find factorizations efficiently; we present run-time examples.

Finally, we count logarithmic signatures for some specific abelian groups and logarithmic signature types.

Encoding. We assume that a mixed radix representation is used for encoding

group elements, i.e. the code length (number of bits required to store a group element) is

$$b_G = \lceil \log_2 |G| \rceil = \left\lceil \log_2(p_1^{k_1} \cdot p_2^{k_2} \cdots p_m^{k_m}) \right\rceil = \left\lceil \sum_{i=1}^{m} (k_i \cdot \log_2 p_i) \right\rceil .$$

Note that this encoding is as compact as possible.

8.1. Rédei's Theorem

One of the most useful observations for factoring in abelian groups is the following theorem of Rédei:

Theorem 8.1. *Let G be an abelian group, and $(A_1, \ldots, A_n) \in \Lambda(G)$ with $0 \in A_i$ and $|A_i| \in \mathbb{P}$ for all $1 \leq i \leq n$. Then at least one of the blocks $A_1, \ldots, A_n$ is a subgroup of G.*

Proof. See [Red65] or [Sza04] (Theorem 1.4.1). □

Exact transversal. Let G be an abelian group, and $\alpha = (A_1, \ldots, A_n) \in \Lambda(G)$ with $0 \in A_i$ and $|A_i| \in \mathbb{P}$ for all $1 \leq i \leq n$. By iterating Rédei's theorem, we see that there exists a permutation of the blocks of α such that the resulting logarithmic signature is exact transversal.

8.2. Structure of Logarithmic Signatures for $\mathbb{Z}_{2^n}$

Lemma 8.2. *Let $x, y \in \mathbb{Z}_{2^n}$. If $\operatorname{ord}(x) > \operatorname{ord}(y)$, then $\operatorname{ord}(x+y) = \operatorname{ord}(x)$.*
In other words, if $\gcd(x, 2^{n-1}) < \gcd(y, 2^{n-1})$, then $\gcd(x+y, 2^{n-1}) = \gcd(x, 2^{n-1})$.

Proof. If $y = 0$, the statements obviously hold. If $x = 0$, neither $\operatorname{ord}(x) > \operatorname{ord}(y)$ nor $\gcd(x, 2^{n-1}) < \gcd(y, 2^{n-1})$ can hold (because 1 is the minimal possible order and $\gcd(y, 2^{n-1})$ cannot be greater than 2^{n-1}). Thus in the following assume $x \neq 0$ and $y \neq 0$.

Write $x = 2^{m_x} \cdot u_x$ with $m_x, u_x \in \mathbb{N}_0$ and $2 \nmid u_x$, and $y = 2^{m_y} \cdot u_y$ with $m_y, u_y \in \mathbb{N}_0$ and $2 \nmid u_y$. We have $2^{m_x} = \gcd(x, 2^{n-1})$ and $2^{m_y} = \gcd(y, 2^{n-1})$. Due to $\gcd(x, 2^{n-1}) < \gcd(y, 2^{n-1})$, we get $2^{m_x} < 2^{m_y}$, i.e. $m_x < m_y$. Therefore,

$$\begin{aligned} \gcd(x+y, 2^{n-1}) &= \gcd(2^{m_x} \cdot u_x + 2^{m_y} \cdot u_y, 2^{n-1}) \\ &= \gcd(2^{m_x} \cdot (u_x + 2^{m_y - m_x} \cdot u_y), 2^{n-1}) \\ &\overset{*}{=} \gcd(2^{m_x}, 2^{n-1}) \overset{**}{=} \gcd(2^{m_x} \cdot u_x, 2^{n-1}) = \gcd(x, 2^{n-1}). \end{aligned}$$

(*) u_x is odd, $2^{m_y - m_x} \cdot u_y$ is even, thus $u_x + 2^{m_y - m_x} \cdot u_y$ is odd.
(**) u_x is odd.
Clearly, $\operatorname{ord}(x) = \frac{2^n}{2^{m_x}}$ (because $\frac{2^n}{2^{m_x}} \cdot x \equiv \frac{2^n}{2^{m_x}} \cdot (2^{m_x} \cdot u_x) \equiv 2^n \cdot u_x \equiv 0 \pmod{2^n}$, and for all $m_x < i \leq n$: $\frac{2^n}{2^i} \cdot x \equiv \frac{2^n}{2^i} \cdot (2^{m_x} \cdot u_x) \equiv \frac{2^n}{2^{i-m_x}} \cdot u_x \not\equiv 0 \pmod{2^n}$) and analogously $\operatorname{ord}(y) = \frac{2^n}{2^{m_y}}$. So, $\operatorname{ord}(x) > \operatorname{ord}(y) \Leftrightarrow \frac{2^n}{2^{m_x}} > \frac{2^n}{2^{m_y}} \Leftrightarrow m_x < m_y$, and thus with the above $\operatorname{ord}(x+y) = \operatorname{ord}(x)$. □

Lemma 8.3. *Let $G = \mathbb{Z}_{2^n}$, $A = \{0, a\}$ with $a \neq 0$, and $B \subseteq G$ canonical with $|A + B| = |A| \cdot |B| = \operatorname{ord}(a)$ and $\operatorname{ord}(a) \geq \max\{\operatorname{ord}(b) \mid b \in B\}$. Then $\operatorname{ord}(a) > \max\{\operatorname{ord}(b) \mid b \in B\}$.*

Proof. With Lemma 8.2, we have $\operatorname{ord}(a) \geq \operatorname{ord}(c)$ for all $c \in A + B$. Together with $|A + B| = \operatorname{ord}(a)$, we get $A + B = \langle a \rangle$. We show by induction that for $0 \leq i < \operatorname{ord}(a)$, $i \cdot a \in B$ holds if and only if $2 \mid i$.

Case $i = 0$: $i \cdot a = 0 \cdot a = 0 \in B$ (due to B canonical).

Case $i > 0$:

- Case $2 \nmid i$. By induction hypothesis, we have $(i-1) \cdot a \in B$. If $i \cdot a \in B$ would hold, we would have two different factorizations for $i \cdot a$ (with respect to $A + B$), namely $0 + i \cdot a$ and $a + (i-1) \cdot a$, which would be a contradiction to $|A + B| = |A| \cdot |B|$. So, $i \cdot a \notin B$.
- Case $2 \mid i$. By induction hypothesis, we have $(i-1) \cdot a \notin B$. If $i \cdot a \notin B$, then $i \cdot a \notin A + B$ (because $i \cdot a$ can neither be written as $0 + i \cdot a$ nor as $a + (i-1) \cdot a$, and 0 and a are the only elements in A), which would be a contradiction to $A + B = \langle a \rangle$. So, $i \cdot a \in B$.

As $\operatorname{ord}(a) > \operatorname{ord}(i \cdot a)$ holds for all $0 \leq i < \operatorname{ord}(a)$ with $2 \mid i$, the asserted statement $\operatorname{ord}(a) > \max\{\operatorname{ord}(b) \mid b \in B\}$ follows. □

Lemma 8.4. *Let $G = \mathbb{Z}_{2^n}$ and $\alpha = (A_1, \ldots, A_n) \in \Lambda(G)$ canonical of type $t(\alpha) = (2, 2, \ldots, 2)$. Then the powers of 2 in the prime factorization of all elements in the logarithmic signature except 0_G are unique, i.e.*

$$\forall\, i \in \{0, 1, \ldots, n-1\}\ \exists!\, g \in (E(\alpha) \setminus \{0\}) : \gcd(g, 2^{n-1}) = 2^i.$$

Equivalently, we have $|\{\operatorname{ord}(A_i[2]) \mid 1 \leq i \leq n\}| = n$.

We give two proofs; first an elementary one and then a shorter one using Rédei's theorem.

Proof. W.l.o.g. assume that the blocks of α are ordered descendingly by the element orders of the second elements, i.e.

$$\operatorname{ord}(A_1[2]) \geq \operatorname{ord}(A_2[2]) \geq \ldots \geq \operatorname{ord}(A_n[2]).$$

Note that $\operatorname{ord}(x) = \frac{2^n}{\gcd(x, 2^{n-1})}$ holds for all $x \in \mathbb{Z}_{2^n} \setminus \{0\}$.

We now show by induction that $\gcd(A_i[2], 2^{n-1}) = 2^{i-1}$ (i.e. $\operatorname{ord}(A_i[2]) = 2^{n-i+1}$) holds for all $1 \leq i \leq n$.

Case $i = 1$. $\mathbb{Z}_{2^n}$ contains elements of order 2^n (namely $1, 3, 5, 7, 9, \ldots$). Thus, $E(\alpha)$ must contain at least one element of order 2^n, otherwise by Lemma 8.2 all generated elements would have orders of 2^{n-1} or less, which would be a contradiction to $\alpha \in \Lambda(G)$. As we assumed $\operatorname{ord}(A_1[2]) \geq \operatorname{ord}(A_2[2]) \geq \ldots \geq \operatorname{ord}(A_n[2])$, we get $\operatorname{ord}(A_1[2]) = 2^n$, i.e. $\gcd(A_1[2], 2^{n-1}) = 1 = 2^{1-1}$.

Case $i > 1$. Let $L_i := A_1 + A_2 + \ldots + A_{i-1}$ and $R_i := A_i + A_{i+1} + \ldots + A_n$ (then $L_i + R_i = \mathbb{Z}_{2^n}$). By induction hypothesis and Lemma 8.2, we get $\operatorname{ord}(g) > 2^{n-i+1}$ for all

$g \in L_i \setminus \{0\}$. Furthermore, with Lemma 8.3, we have $\max\{\text{ord}(g) \mid g \in R_i\} \leq 2^{n-i+1}$. $\mathbb{Z}_{2^n}$ contains elements of order 2^{n-i+1} (namely $1 \cdot 2^{i-1}, 3 \cdot 2^{i-1}, 5 \cdot 2^{i-1}, \ldots$), consequently $E((A_i, A_{i+1}, \ldots, A_n))$ must contain at least one element of order 2^{n-i+1}, otherwise $\text{ord}(g) < 2^{n-i+1}$ would hold for all $g \in R_i$, which would be a contradiction to $\alpha \in \Lambda(G)$ (because α would not generate any element of order 2^{n-i+1}; we would have $\text{ord}(g) < 2^{n-i+1}$ for all $g \in R_i$ and $\text{ord}(g) > 2^{n-i+1}$ for all $g \in (L_i \setminus \{0\}) + R_i$). As we assumed $\text{ord}(A_1[2]) \geq \text{ord}(A_2[2]) \geq \ldots \geq \text{ord}(A_n[2])$, we get $\text{ord}(A_i[2]) = 2^{n-i+1}$, i.e. $\gcd(A_i[2], 2^{n-1}) = 2^{i-1}$. □

Now the shorter proof using Rédei's theorem:

Proof. According to Rédei's theorem, at least one block of α must be a subgroup. As every block contains exactly two elements, a subgroup block must be of the form $(0, u)$ with $u = -u$. Hence $u = 2^{n-1}$, because this is the only self-inverse non-0 element in $\mathbb{Z}_{2^n}$.

Let $G = R + \{0, u\}$ (i.e. R is the sum of all other blocks). For every $g \in G$, we have either $g \in R$ or $g + u \in R$. Thus the mapping $R \to \mathbb{Z}_{2^{n-1}} : g \mapsto g \bmod 2^{n-1}$ is bijective. When interpreting the elements of R in $\mathbb{Z}_{2^{n-1}}$, the blocks of R form a logarithmic signature of $\mathbb{Z}_{2^{n-1}}$.

Apply the argument recursively on R and the statement of Lemma 8.4 follows. □

8.3. Factoring in $\mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^k$

Algorithm 8.5. Let $H = \mathbb{Z}_{2^{\widetilde{m}}} \oplus \mathbb{Z}_2^{\widetilde{k}}$ for some $\widetilde{m} \in \mathbb{N}$, $\widetilde{k} \in \mathbb{N}_0$. Let $\beta \in \Lambda(H)$ canonical of type $t(\beta) = (2, 2, \ldots, 2)$ and $h \in H$.

The following recursive function finds the factorization of h with respect to β, when calling `FactorizeZ2mZ2k`$(\beta, h, \widetilde{m}, \widetilde{k})$:

Function `FactorizeZ2mZ2k`(LogSig α, GroupElement g, UInt m, UInt k) : List<UInt>

1. If $|\alpha| = 1$: find g in $\alpha[1]$ and `return` its index (one item in a list).
2. Let $n \leftarrow m + k$.
3. Let i be the index of the first block in α being a subgroup (i.e. with $\text{ord}(\alpha[i][2]) = 2$).
   Let $(u, v_1, \ldots, v_k) \leftarrow \alpha[i][2]$.
4. If $k \geq 1$ and at least one of the $v_1, \ldots, v_k$ is 1:
   a) Let $j \leftarrow \min\{z \in \{1, \ldots, k\} \mid v_z = 1\}$.
   b) Let $f : \mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^k \to \mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^{k-1} : (x, y_1, \ldots, y_k) \mapsto (x + y_j \cdot u, y_1 + y_j \cdot v_1, \ldots, y_{i-1} + y_j \cdot v_{i-1}, y_{i+1} + y_j \cdot v_{i+1}, \ldots, y_k + y_j \cdot v_k)$.
   Let $\alpha' \leftarrow \begin{pmatrix}(0,\ldots,0)\\ f(\alpha[1][2])\end{pmatrix} \cdots \begin{pmatrix}(0,\ldots,0)\\ f(\alpha[i-1][2])\end{pmatrix} \begin{pmatrix}(0,\ldots,0)\\ f(\alpha[i+1][2])\end{pmatrix} \cdots \begin{pmatrix}(0,\ldots,0)\\ f(\alpha[n][2])\end{pmatrix}$.
   c) Let $r \leftarrow$ `FactorizeZ2mZ2k`$(\alpha', f(g), m, k-1)$.

else:

a) Let $f : \mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^k \to \mathbb{Z}_{2^{m-1}} \oplus \mathbb{Z}_2^k : (x, y_1, \ldots, y_k) \mapsto (x \bmod 2^{m-1}, y_1, \ldots, y_k)$.
Let $\alpha' \leftarrow \binom{(0,\ldots,0)}{f(\alpha[1][2])} \cdots \binom{(0,\ldots,0)}{f(\alpha[i-1][2])} \binom{(0,\ldots,0)}{f(\alpha[i+1][2])} \cdots \binom{(0,\ldots,0)}{f(\alpha[n][2])}$.

b) Let $r \leftarrow$ `FactorizeZ2mZ2k`$(\alpha', f(g), m-1, k)$.

5. Insert 1 at position i in r.

6. If $\alpha[1][r[1]] + \ldots + \alpha[n][r[n]] = g$: `return` r.
Let $r[i] \leftarrow 2$ and `return` r.

Proof. Let $G := \mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^k$ and $n := m + k$. The algorithm realizes the idea of Theorem 7.6.

According to Rédei's theorem, at least one of the blocks of $\alpha = (A_1, A_2, \ldots, A_n)$ is a subgroup of G. Let i be the index of the first block being a subgroup. We have $A_i = ((0, \ldots, 0), s)$ for an $s \in G \setminus \{(0, \ldots, 0)\}$. Write $s = (u, v_1, v_2, \ldots, v_k)$ with $u \in \mathbb{Z}_{2^m}$ and $v_1, \ldots, v_k \in \mathbb{Z}_2$.

Define the projection $\gamma_j : G \to \mathbb{Z}_2 : (x, y_1, y_2, \ldots, y_k) \mapsto y_j$.

As A_i is a subgroup, $s = -s$ and consequently $u \in \{0, 2^{m-1}\}$ (because only 0 and 2^{m-1} are self-inverse in $\mathbb{Z}_{2^m}$). We distinguish two cases:

- At least one of the $v_1, \ldots, v_k$ is 1. Let v_j be the first one of those (i.e. $v_j = 1$). Now, for all other blocks A_t with $t \neq i$, we add s to $A_t[2]$, if $\gamma_j(A_t[2]) = 1$. These additions are selective shifts and therefore result in another logarithmic signature. In the resulting logarithmic signature, we have $\gamma_j(A_i[2]) = 1$ and $\gamma_j(A_t[2]) = 0$ for all $t \neq i$. Thus, by removing the block A_i and the $(j+1)$th component (note that the one for $\mathbb{Z}_{2^m}$ is the first) from all elements (f performs the addition and removal at once), we obtain a logarithmic signature α' for the group

$$\mathbb{Z}_{2^m} \oplus \underbrace{\mathbb{Z}_2 \oplus \ldots \oplus \mathbb{Z}_2}_{j-1} \oplus \underbrace{\mathbb{Z}_2 \oplus \ldots \oplus \mathbb{Z}_2}_{k-j} = \mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^{k-1}.$$

- All of the $v_1, \ldots, v_k$ are 0. Then $u = 2^{m-1}$ (because $u = 0$ would result in $s = (0, \ldots, 0)$, a contradiction). Let $M := A_1 + \ldots + A_{i-1} + A_{i+1} + \ldots + A_n$ and $G' := \mathbb{Z}_{2^{m-1}} \oplus \mathbb{Z}_2^k$. For each $g \in G$, we have either $g \in M$ or $g + s \in M$. Thus, f is bijective from M to G', and therefore $\alpha' = (f(A_1), \ldots, f(A_{i-1}), f(A_{i+1}), \ldots, f(A_n)) \in \Lambda(G')$.

In order to compute the factorization of g with respect to α, first factor $f(g)$ with respect to α', select the corresponding indices in α and complete the factorization by choosing the correct element in A_i (only two possibilities). □

Before giving a few examples illustrating Algorithm 8.5, we first want to have an in-depth look at its run-time. Although the run-time clearly is polynomial in the input size, we want to explicitly find an asymptotic bound for the run-time.

Proposition 8.6. *Let $G = \mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^k$ for some $m \in \mathbb{N}$, $k \in \mathbb{N}_0$, $n := m + k$, and $\alpha = (A_1, A_2, \ldots, A_n) \in \Lambda(G)$ canonical of type $t(\alpha) = (2, 2, \ldots, 2)$.*

Algorithm 8.5 performs $O(n^2)$ group element operations/tests.

The input size to Algorithm 8.5 is $w := 2n^2 + n + 2\lceil \log_2(n+1) \rceil$ bits, and the run-time is in $O(w^{\frac{3}{2}})$.

Thus, α is tame.

Proof. The code length (number of bits required to store a group element) is $b_G = n$. The input to the algorithm is a logarithmic signature (requiring $\mathcal{S}(G, \alpha) = b_G \cdot \ell(\alpha)$ bits), an element to factor (requiring b_G bits), and the numbers m and k (each requiring $\lceil \log_2(n+1) \rceil$ bits). In total, the input size is $w = b_G \cdot \ell(\alpha) + b_G + 2\lceil \log_2(n+1) \rceil = n \cdot 2n + n + 2\lceil \log_2(n+1) \rceil = 2n^2 + n + 2\lceil \log_2(n+1) \rceil$ bits.

Let $g \in G$ be the element to be factored.

If $|\alpha| = 1$ (i.e. $|G| = 2$), one group element test is sufficient (test whether $g = (0, \ldots, 0)$).

Thus now assume $n \geq 2$, i.e. $|G| \geq 4$. The first step is to search the subgroup block. As α is canonical, regarding only the second element in each block is sufficient. We need at most $n - 1$ group element tests in order to determine the index of the subgroup block (for $1 \leq i \leq n - 1$ test whether $A_i[2]$ is self-inverse; after $n - 1$ unsuccessful tests, we know that $A_n[2]$ is the self-inverse element).

In both cases that can occur now, we construct a smaller logarithmic signature for a smaller group (and call us recursively). We need one group element test to decide which case occurs.

- If at least one of the $v_1, \ldots, v_k$ is 1, e.g. v_j, create a new logarithmic signature consisting of $n - 1$ blocks, containing the identity of $G' := \mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^{k-1}$ as the first element in each block ($n - 1$ group operations). Using $n - 1$ group element tests, we compute $\gamma_j(A_t[2])$ for all other blocks (that do not contain s), and depending on the results we possibly add s to them (at most $n - 1$ group operations). Finally, we remove the $(j + 1)$th component from the $n - 1$ elements to obtain G' elements, which requires $n - 1$ group operations. Furthermore, we need to reduce g, which analogously requires at most 3 group element tests/operations.

- If all of the $v_1, \ldots, v_k$ are 0, create a new logarithmic signature consisting of $n - 1$ blocks, containing the identity of $G' := \mathbb{Z}_{2^{m-1}} \oplus \mathbb{Z}_2^k$ as the first element in each block ($n - 1$ group operations). Using $n - 1$ group element operations, we perform the modulo computations in the first components to obtain G' elements (note that a mod 2^{m-1} computation can be realized by simply cutting off the leftmost bit). Furthermore, we need to reduce g, which analogously requires at most one group element operation.

When the recursion returns, we compute the sum of the elements at the returned factorization indices in α and select the appropriate element in the subgroup block (if the sum is g, $(0, \ldots, 0)$ needs to be selected in the subgroup block, otherwise s); this requires $(n - 2) + 1 = n - 1$ group element operations/tests. There is no need to undo the transformation, because the original logarithmic signature is still in memory (we created a completely new logarithmic signature for the recursive call).

Let $t(n)$ be the time that a group element operation/test requires when the group element is encoded using n bits. As $G = \mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^k$, we can assume that $t(n) \in O(n)$.

So, the total run-time $T(n)$ of the algorithm with respect to n is bounded by

$$\begin{aligned} T(n) &= ((n-1) + 1 + \max\{4(n-1)+3,\, 2(n-1)+1\}) \cdot t(n) + T(n-1) + (n-1) \cdot t(n) \\ &= T(n-1) + (6(n-1)+4) \cdot t(n) \end{aligned}$$

for $n \geq 2$, $T(1) = t(1)$, and thus $T(n) \in O(n^3)$.

However, the run-time of algorithms is usually expressed with respect to the input size w. With respect to w, the run-time of the algorithm is in $O(\sqrt{w}^3) = O(w^{\frac{3}{2}})$.

Note that there are many possibilities to further optimize the implementation of the factorization algorithm. For example, copying the identity elements in the blocks is rather unnecessary; a subroutine could be written that (recursively) works with the second elements of the original input only. □

Example 8.7. We want to factor $g := (13, 0, 1, 0)$ with respect to the following logarithmic signature of $G := \mathbb{Z}_{2^4} \oplus \mathbb{Z}_2^3$:

$$\alpha := \tbinom{(0,0,0,0)}{(5,1,1,1)}\tbinom{(0,0,0,0)}{(15,0,1,1)}\tbinom{(0,0,0,0)}{(10,0,1,0)}\tbinom{(0,0,0,0)}{(12,1,1,1)}\tbinom{(0,0,0,0)}{(12,0,1,0)}\tbinom{(0,0,0,0)}{(8,0,0,0)}\tbinom{(0,0,0,0)}{(8,1,1,1)}.$$

A_6 is the first block being a subgroup, and all $v_j = 0$ (i.e. the second case of Algorithm 8.5 occurs). So, remove A_6 and reduce the first component of the elements in all other blocks mod 8 to get a logarithmic signature for $\mathbb{Z}_{2^3} \oplus \mathbb{Z}_2^3$:

$$\alpha_1 := \tbinom{(0,0,0,0)}{(5,1,1,1)}\tbinom{(0,0,0,0)}{(7,0,1,1)}\tbinom{(0,0,0,0)}{(2,0,1,0)}\tbinom{(0,0,0,0)}{(4,1,1,1)}\tbinom{(0,0,0,0)}{(4,0,1,0)}\tbinom{(0,0,0,0)}{(0,1,1,1)}.$$

Now, the fourth block is a subgroup (recall that we are now working in the reduced group). Here, $v_1 = 1$, i.e. the first case of Algorithm 8.5 occurs. By adding $(4, 1, 1, 1)$ to the others with $\gamma_1(A_t[2]) = 1$, removing the block containing $(4, 1, 1, 1)$ and the second component of the elements in all other blocks, we get the following logarithmic signature for $\mathbb{Z}_{2^3} \oplus \mathbb{Z}_2^2$:

$$\alpha_2 := \tbinom{(0,0,0)}{(1,0,0)}\tbinom{(0,0,0)}{(7,1,1)}\tbinom{(0,0,0)}{(2,1,0)}\tbinom{(0,0,0)}{(4,1,0)}\tbinom{(0,0,0)}{(4,0,0)}.$$

Continuing this way, we get the following logarithmic signatures:

$$\begin{aligned} \alpha_3 &:= \tbinom{(0,0)}{(1,0)}\tbinom{(0,0)}{(3,1)}\tbinom{(0,0)}{(6,0)}\tbinom{(0,0)}{(4,0)} \text{ (for } \mathbb{Z}_{2^3} \oplus \mathbb{Z}_2^1), \\ \alpha_4 &:= \tbinom{(0,0)}{(1,0)}\tbinom{(0,0)}{(3,1)}\tbinom{(0,0)}{(2,0)} \text{ (for } \mathbb{Z}_{2^2} \oplus \mathbb{Z}_2^1), \\ \alpha_5 &:= \tbinom{(0,0)}{(1,0)}\tbinom{(0,0)}{(1,1)} \text{ (for } \mathbb{Z}_2 \oplus \mathbb{Z}_2^1), \\ \alpha_6 &:= \tbinom{(0,0)}{(0,1)} \text{ (for } \{0\} \oplus \mathbb{Z}_2^1). \end{aligned}$$

Now to the factorization of $g = (13, 0, 1, 0)$. In order to compute its factorization in α, we need to know the factorization of $(13 \bmod 8, 0, 1, 0) = (5, 0, 1, 0)$ in α_1. In order to get its factorization in α_1, we need to compute $(5, 1, 0)$ in α_2. Observe that there is a 1 in the second component, therefore compute $(5, 1, 0) + (4, 1, 0) = (1, 0, 0)$ and conclude that we need to factor $(1, 0)$ in α_3. Continuing like this, we need to factor $(1, 0)$ in α_4 and α_5, and $(0, 0)$ in α_6.

Knowing that we need to factor $(0,0)$ in α_6 (which is easy), we can deduce the factorization of $(1,0)$ in α_5 by going up again (from α_5 to α_6, we have reduced the first component).

$$\alpha_6 = \left(\boxed{\tbinom{(0,0)}{(0,1)}}\right),$$
$$\alpha_5 = \left(\boxed{\tbinom{(0,0)}{(1,0)}}\right)\left(\boxed{\tbinom{(0,0)}{(1,1)}}\right).$$

From α_4 to α_5 and from α_3 to α_4, we have reduced the first component, so we get:

$$\alpha_4 = \left(\boxed{\tbinom{(0,0)}{(1,0)}}\right)\left(\boxed{\tbinom{(0,0)}{(3,1)}}\right)\left(\boxed{\tbinom{(0,0)}{(2,0)}}\right),$$
$$\alpha_3 = \left(\boxed{\tbinom{(0,0)}{(1,0)}}\right)\left(\boxed{\tbinom{(0,0)}{(3,1)}}\right)\left(\boxed{\tbinom{(0,0)}{(6,0)}}\right)\left(\boxed{\tbinom{(0,0)}{(4,0)}}\right).$$

In the step from α_2 to α_3, the second component was removed. We already know that we need to factor $(5,1,0)$ in α_2, and the only way to achieve this is by selecting

$$\alpha_2 = \left(\boxed{\tbinom{(0,0,0)}{(1,0,0)}}\right)\left(\boxed{\tbinom{(0,0,0)}{(7,1,1)}}\right)\left(\boxed{\tbinom{(0,0,0)}{(2,1,0)}}\right)\left(\boxed{\tbinom{(0,0,0)}{(4,1,0)}}\right)\left(\boxed{\tbinom{(0,0,0)}{(4,0,0)}}\right).$$

We already know we need $(5,0,1,0)$ in α_1. As the second component is 0, we simply get:

$$\alpha_1 = \left(\boxed{\tbinom{(0,0,0,0)}{(5,1,1,1)}}\right)\left(\boxed{\tbinom{(0,0,0,0)}{(7,0,1,1)}}\right)\left(\boxed{\tbinom{(0,0,0,0)}{(2,0,1,0)}}\right)\left(\boxed{\tbinom{(0,0,0,0)}{(4,1,1,1)}}\right)\left(\boxed{\tbinom{(0,0,0,0)}{(4,0,1,0)}}\right)\left(\boxed{\tbinom{(0,0,0,0)}{(0,1,1,1)}}\right).$$

Computing the unreduced sum:

$$\alpha = \left(\boxed{\tbinom{(0,0,0,0)}{(5,1,1,1)}}\right)\left(\boxed{\tbinom{(0,0,0,0)}{(15,0,1,1)}}\right)\left(\boxed{\tbinom{(0,0,0,0)}{(10,0,1,0)}}\right)\left(\boxed{\tbinom{(0,0,0,0)}{(12,1,1,1)}}\right)\left(\boxed{\tbinom{(0,0,0,0)}{(12,0,1,0)}}\right)\left(\boxed{\tbinom{(0,0,0,0)}{(8,0,0,0)}}\right)\left(\boxed{\tbinom{(0,0,0,0)}{(8,1,1,1)}}\right).$$

This indeed is the factorization of $g = (13,0,1,0)$.

Example 8.8. Another example how to find factorizations using Algorithm 8.5; this time without comments (see previous example for detailed explanations of all steps).

We want to factor $g := (11,1,1,0)$ in the following logarithmic signature of $G := \mathbb{Z}_{2^4} \oplus \mathbb{Z}_2^3$:

$$\alpha := \left(\tbinom{(0,0,0,0)}{(0,0,1,1)}\right)\left(\tbinom{(0,0,0,0)}{(11,1,0,1)}\right)\left(\tbinom{(0,0,0,0)}{(6,1,1,1)}\right)\left(\tbinom{(0,0,0,0)}{(12,0,1,1)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,0,1)}\right)\left(\tbinom{(0,0,0,0)}{(8,1,1,1)}\right)\left(\tbinom{(0,0,0,0)}{(8,1,0,1)}\right),$$
$$g = (11,1,1,0),$$
$$\alpha_1 := \left(\tbinom{(0,0,0)}{(11,1,1)}\right)\left(\tbinom{(0,0,0)}{(6,1,0)}\right)\left(\tbinom{(0,0,0)}{(12,0,0)}\right)\left(\tbinom{(0,0,0)}{(8,0,1)}\right)\left(\tbinom{(0,0,0)}{(8,1,0)}\right)\left(\tbinom{(0,0,0)}{(8,1,1)}\right),$$
$$g_1 := (11,1,1),$$
$$\alpha_2 := \left(\tbinom{(0,0)}{(3,1)}\right)\left(\tbinom{(0,0)}{(6,1)}\right)\left(\tbinom{(0,0)}{(12,0)}\right)\left(\tbinom{(0,0)}{(8,1)}\right)\left(\tbinom{(0,0)}{(0,1)}\right),$$
$$g_2 := (3,1),$$
$$\alpha_3 := \left(\tbinom{(0)}{(11)}\right)\left(\tbinom{(0)}{(14)}\right)\left(\tbinom{(0)}{(12)}\right)\left(\tbinom{(0)}{(8)}\right),$$
$$g_3 := (11),$$
$$\alpha_4 := \left(\tbinom{(0)}{(3)}\right)\left(\tbinom{(0)}{(6)}\right)\left(\tbinom{(0)}{(4)}\right),$$

$$g_4 := (3),$$
$$\alpha_5 := \binom{(0)}{(3)}\binom{(0)}{(2)},$$
$$g_5 := (3),$$
$$\alpha_6 := \binom{(0)}{(1)},$$
$$g_6 := (1).$$

Going up again:

$$\alpha_6 = \binom{(0)}{\boxed{(1)}},$$
$$\alpha_5 = \binom{(0)}{\boxed{(3)}}\binom{\boxed{(0)}}{(2)},$$
$$\alpha_4 = \binom{(0)}{\boxed{(3)}}\binom{\boxed{(0)}}{(6)}\binom{\boxed{(0)}}{(4)},$$
$$\alpha_3 = \binom{(0)}{\boxed{(11)}}\binom{\boxed{(0)}}{(14)}\binom{\boxed{(0)}}{(12)}\binom{\boxed{(0)}}{(8)},$$
$$\alpha_2 = \binom{(0,0)}{\boxed{(3,1)}}\binom{\boxed{(0,0)}}{(6,1)}\binom{\boxed{(0,0)}}{(12,0)}\binom{\boxed{(0,0)}}{(8,1)}\binom{\boxed{(0,0)}}{(0,1)},$$
$$\alpha_1 = \binom{(0,0,0)}{\boxed{(11,1,1)}}\binom{\boxed{(0,0,0)}}{(6,1,0)}\binom{\boxed{(0,0,0)}}{(12,0,0)}\binom{\boxed{(0,0,0)}}{(8,0,1)}\binom{\boxed{(0,0,0)}}{(8,1,0)}\binom{\boxed{(0,0,0)}}{(8,1,1)},$$
$$\alpha = \binom{(0,0,0,0)}{\boxed{(0,0,1,1)}}\binom{(0,0,0,0)}{\boxed{(11,1,0,1)}}\binom{\boxed{(0,0,0,0)}}{(6,1,1,1)}\binom{\boxed{(0,0,0,0)}}{(12,0,1,1)}\binom{\boxed{(0,0,0,0)}}{(8,0,0,1)}\binom{\boxed{(0,0,0,0)}}{(8,1,1,1)}\binom{\boxed{(0,0,0,0)}}{(8,1,0,1)}.$$

Indeed, this is the factorization of $g = (11, 1, 1, 0)$.

Example 8.9. Another example; this time with $g := (3, 1, 0, 1)$ in the following logarithmic signature of $G := \mathbb{Z}_{2^4} \oplus \mathbb{Z}_2^3$:

$$\alpha := \binom{(0,0,0,0)}{(0,0,1,1)}\binom{(0,0,0,0)}{(3,0,1,1)}\binom{(0,0,0,0)}{(13,1,1,1)}\binom{(0,0,0,0)}{(14,1,0,1)}\binom{(0,0,0,0)}{(4,0,0,0)}\binom{(0,0,0,0)}{(4,1,1,0)}\binom{(0,0,0,0)}{(8,0,1,1)},$$
$$g = (3, 1, 0, 1),$$
$$\alpha_1 := \binom{(0,0,0)}{(3,0,0)}\binom{(0,0,0)}{(13,1,0)}\binom{(0,0,0)}{(14,1,1)}\binom{(0,0,0)}{(4,0,0)}\binom{(0,0,0)}{(4,1,1)}\binom{(0,0,0)}{(8,0,0)},$$
$$g_1 := (3, 1, 1),$$
$$\alpha_2 := \binom{(0,0,0)}{(3,0,0)}\binom{(0,0,0)}{(5,1,0)}\binom{(0,0,0)}{(6,1,1)}\binom{(0,0,0)}{(4,0,0)}\binom{(0,0,0)}{(4,1,1)},$$
$$g_2 := (3, 1, 1),$$
$$\alpha_3 := \binom{(0,0,0)}{(3,0,0)}\binom{(0,0,0)}{(1,1,0)}\binom{(0,0,0)}{(2,1,1)}\binom{(0,0,0)}{(0,1,1)},$$
$$g_3 := (3, 1, 1),$$
$$\alpha_4 := \binom{(0,0)}{(3,0)}\binom{(0,0)}{(3,1)}\binom{(0,0)}{(2,0)},$$
$$g_4 := (1, 0),$$
$$\alpha_5 := \binom{(0,0)}{(1,0)}\binom{(0,0)}{(1,1)},$$
$$g_5 := (1, 0),$$
$$\alpha_6 := \binom{(0,0)}{(0,1)},$$
$$g_6 := (0, 0).$$

Going up again:

$$\alpha_6 = \left(\boxed{\begin{smallmatrix}(0,0)\\(0,1)\end{smallmatrix}}\right),$$
$$\alpha_5 = \left(\begin{smallmatrix}(0,0)\\\boxed{(1,0)}\end{smallmatrix}\right)\left(\boxed{\begin{smallmatrix}(0,0)\\(1,1)\end{smallmatrix}}\right),$$
$$\alpha_4 = \left(\begin{smallmatrix}(0,0)\\\boxed{(3,0)}\end{smallmatrix}\right)\left(\boxed{\begin{smallmatrix}(0,0)\\(3,1)\end{smallmatrix}}\right)\left(\begin{smallmatrix}(0,0)\\\boxed{(2,0)}\end{smallmatrix}\right),$$
$$\alpha_3 = \left(\begin{smallmatrix}(0,0,0)\\\boxed{(3,0,0)}\end{smallmatrix}\right)\left(\boxed{\begin{smallmatrix}(0,0,0)\\(1,1,0)\end{smallmatrix}}\right)\left(\boxed{\begin{smallmatrix}(0,0,0)\\(2,1,1)\end{smallmatrix}}\right)\left(\begin{smallmatrix}(0,0,0)\\\boxed{(0,1,1)}\end{smallmatrix}\right),$$
$$\alpha_2 = \left(\begin{smallmatrix}(0,0,0)\\\boxed{(3,0,0)}\end{smallmatrix}\right)\left(\boxed{\begin{smallmatrix}(0,0,0)\\(5,1,0)\end{smallmatrix}}\right)\left(\boxed{\begin{smallmatrix}(0,0,0)\\(6,1,1)\end{smallmatrix}}\right)\left(\begin{smallmatrix}(0,0,0)\\\boxed{(4,0,0)}\end{smallmatrix}\right)\left(\begin{smallmatrix}(0,0,0)\\\boxed{(4,1,1)}\end{smallmatrix}\right),$$
$$\alpha_1 = \left(\begin{smallmatrix}(0,0,0)\\\boxed{(3,0,0)}\end{smallmatrix}\right)\left(\boxed{\begin{smallmatrix}(0,0,0)\\(13,1,0)\end{smallmatrix}}\right)\left(\boxed{\begin{smallmatrix}(0,0,0)\\(14,1,1)\end{smallmatrix}}\right)\left(\begin{smallmatrix}(0,0,0)\\\boxed{(4,0,0)}\end{smallmatrix}\right)\left(\begin{smallmatrix}(0,0,0)\\\boxed{(4,1,1)}\end{smallmatrix}\right)\left(\begin{smallmatrix}(0,0,0)\\\boxed{(8,0,0)}\end{smallmatrix}\right),$$
$$\alpha = \left(\begin{smallmatrix}(0,0,0,0)\\\boxed{(0,0,1,1)}\end{smallmatrix}\right), \left(\begin{smallmatrix}(0,0,0,0)\\\boxed{(3,0,1,1)}\end{smallmatrix}\right), \left(\boxed{\begin{smallmatrix}(0,0,0,0)\\(13,1,1,1)\end{smallmatrix}}\right), \left(\boxed{\begin{smallmatrix}(0,0,0,0)\\(14,1,0,1)\end{smallmatrix}}\right), \left(\begin{smallmatrix}(0,0,0,0)\\\boxed{(4,0,0,0)}\end{smallmatrix}\right), \left(\begin{smallmatrix}(0,0,0,0)\\\boxed{(4,1,1,0)}\end{smallmatrix}\right), \left(\begin{smallmatrix}(0,0,0,0)\\\boxed{(8,0,1,1)}\end{smallmatrix}\right).$$

Success, this is the factorization of $g = (3, 1, 0, 1)$.

Proposition 8.10. *Let $G = \mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^k$ for some $m \in \mathbb{N}$, $k \in \mathbb{N}_0$, and α a canonical sequence of $m + k$ blocks of size 2.*

Testing whether α is a logarithmic signature for G is possible in polynomial time.

Proof. The idea is to apply several transformations, which are described below, to α. If α is a logarithmic signature, each of these transformations will result in another logarithmic signature. Conversely, if α is not a logarithmic signature, the transformations will not result in one.

At the end, α is a logarithmic signature if and only if the final transformed block sequence is $(B_1, B_2, \ldots, B_{m+k})$ with $B_i = \{(0, \ldots, 0), (2^{i-1}, 0, 0, \ldots, 0)\}$ for $1 \le i \le m$ and $B_i = \{(0, \ldots, 0), (0, \underbrace{0, \ldots, 0}_{i-(m+1)}, 1, \underbrace{0, \ldots, 0}_{m+k-i})\}$ for $m + 1 \le i \le m + k$.

We now describe the transformation process. First, apply the transformations of Algorithm 8.5 to α, but without removing blocks and without shortening the vectors. In the resulting logarithmic signature there are exactly m vectors of the form $(u, 0, 0, \ldots, 0)$ with $u \in \mathbb{Z}_{2^m}$, which obviously are generating $R := \mathbb{Z}_{2^m} \oplus \{0\}^k$. Thus we can set the first component in all other vectors to zero. These other vectors are then generating $V := \{0\} \oplus \mathbb{Z}_2^k$ (a vector space). By using standard linear column transformations, we achieve that each of those vectors has a 1 at exactly one position. Normalize the blocks generating R by choosing the smallest possible representatives: replace every u by the smallest element in $\langle u \rangle$ (this is possible efficiently: simply replace every u by u AND $((2^m - 1)$ XOR $(u - 1))$, with bitwise AND and XOR). Finally, sort the blocks (the vectors generating R to the front and ordered lexicographically, subsequently the vectors generating V ordered like above). □

Remark 8.11. Note that the transformations used in the proof of Proposition 8.10 are not directly invertible when considering factorizations. For example, the step of zeroing out the first component clearly results in another logarithmic signature, but knowing a

factorization in the transformed signature does not allow us to derive a factorization in the original one immediately.

In contrast, applying the methods of Algorithm 8.5 allows us to recursively compute factorizations easily.

Example 8.12. Let $G := \mathbb{Z}_{2^4} \oplus \mathbb{Z}_2^3$ and

$$\alpha := \left(\tbinom{(0,0,0,0)}{(0,0,0,1)}\right)\left(\tbinom{(0,0,0,0)}{(11,0,1,1)}\right)\left(\tbinom{(0,0,0,0)}{(10,1,1,1)}\right)\left(\tbinom{(0,0,0,0)}{(4,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,1,1,1)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,0,0)}\right).$$

We want to test whether α is a logarithmic signature for G. So, we first apply the transformations of Algorithm 8.5 without removing blocks and without shortening the vectors:

$$\begin{aligned}
&\rightsquigarrow \left(\tbinom{(0,0,0,0)}{(0,0,0,1)}\right)\left(\tbinom{(0,0,0,0)}{(11,0,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(10,1,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(4,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,1,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,0,0)}\right),\\
&\rightsquigarrow \left(\tbinom{(0,0,0,0)}{(0,0,0,1)}\right)\left(\tbinom{(0,0,0,0)}{(11,0,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(2,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(4,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,1,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,0,0)}\right),\\
&\rightsquigarrow \left(\tbinom{(0,0,0,0)}{(0,0,0,1)}\right)\left(\tbinom{(0,0,0,0)}{(3,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(2,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(4,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,1,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,0,0)}\right),\\
&\rightsquigarrow \left(\tbinom{(0,0,0,0)}{(0,0,0,1)}\right)\left(\tbinom{(0,0,0,0)}{(1,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(2,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(4,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,1,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,0,0)}\right).
\end{aligned}$$

Now we cancel out the first component of all vectors with a 1 in the second or higher component:

$$\rightsquigarrow \left(\tbinom{(0,0,0,0)}{(0,0,0,1)}\right)\left(\tbinom{(0,0,0,0)}{(1,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(2,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(4,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(0,1,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(0,0,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,0,0)}\right).$$

Using linear column transformations (here: just add the 6th vector onto the 5th), we get:

$$\rightsquigarrow \left(\tbinom{(0,0,0,0)}{(0,0,0,1)}\right)\left(\tbinom{(0,0,0,0)}{(1,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(2,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(4,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(0,1,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(0,0,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,0,0)}\right).$$

By sorting the vectors we obtain the final result:

$$\rightsquigarrow \left(\tbinom{(0,0,0,0)}{(1,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(2,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(4,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(8,0,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(0,1,0,0)}\right)\left(\tbinom{(0,0,0,0)}{(0,0,1,0)}\right)\left(\tbinom{(0,0,0,0)}{(0,0,0,1)}\right).$$

This is the logarithmic signature described in the proof of Proposition 8.10, so α indeed is a logarithmic signature.

8.4. Factoring in Logarithmic Signatures with Blocks of Prime Size

Let G be an abelian group (represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$) and $\alpha = (A_1, A_2, \ldots, A_n) \in \Lambda(G)$ canonical with $|A_i| \in \mathbb{P}$ for all $1 \leq i \leq n$. We now design an algorithm (Algorithm 8.16) that computes factorizations with respect to α using $O(q^3 \cdot m \cdot n^3 \cdot \ell(\alpha))$ group operations, where $q := \max\{|A_i| \mid 1 \leq i \leq n\}$. Thus, α is tame.

This is one of the main results in our work. Note that it supersedes Algorithm 8.5.

The core idea for the algorithm is the following. According to Rédei's theorem

(Theorem 8.1), at least one block A_i in α is a subgroup. Subgroups of abelian groups are normal, thus the factor group G/A_i is well-defined. By applying Theorem 7.6 recursively, we can factorize every group element efficiently.

For this, we must be able to efficiently represent and perform computations in G/N (where N is a subgroup; $N = A_i$ on the topmost recursion level). It might not be immediately clear that this is possible. In fact, if elements in G/N would be represented by sets of elements (like in the theoretic definition of a normal subgroup), then the procedure would require exponential time, because the sets grow exponentially while descending in the recursion. Also, converting elements to permutations and using algorithms based on permutations (which many computer algebra systems use) might not be appropriate here, because the space required to store the permutations might be exponential (e.g. the permutation representation of elements in a cyclic group $\mathbb{Z}_{2^n}$ has degree 2^n; in the compact element representation that we assume in this section for the input an element requires $\lceil \log_2 2^n \rceil = n$ bits, but when converting such an element to a permutation and storing it as a list of images or storing its cycle notation, we require $2^n \cdot n$ bits).

However, it is possible to efficiently compute in G/N in this case here. In the following, we first show how several specific computations related to G/N can be carried out efficiently, and subsequently we describe the factorization algorithm.

Algorithm 8.13. Let G be an abelian group (represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$) and $N := A_1 + A_2 + \ldots + A_r$ with $A_1 \leq G$, $A_1 + A_2 \leq G$, $A_1 + A_2 + A_3 \leq G$, …, $N \leq G$. Let $x, y \in G$.

We present two algorithms that together test whether $x + N = y + N$ holds.

We expect the blocks $A_1, \ldots, A_r$ to be given as a block sequence $\alpha := (A_1, \ldots, A_r)$, and x and y as elements of G. The set N (containing $|A_1| \cdot |A_2| \cdots |A_r|$ elements) is *not* part of the input; it is only used for notation.

The first algorithm computes a list $\mathfrak{C}$ of m group elements. Using $\mathfrak{C}$, the second algorithm tests whether $x + N = y + N$ holds.

Computing $\mathfrak{C}$ is independent of x and y, i.e. when multiple elements are being tested, computing $\mathfrak{C}$ once is sufficient. In the LogSig utility (Chapter 12), we have implemented this performance optimization.

$\mathfrak{C}$ is computed by the following algorithm:

1. Let $\alpha' = (A'_1, \ldots, A'_r) \leftarrow \alpha$.
2. Let $\mathfrak{C} \leftarrow ((0, \ldots, 0), \ldots, (0, \ldots, 0)) \in G^m$.
3. For $c \leftarrow 1$ to m:
   - For $i \leftarrow 1$ to r:
     a) Let $S \leftarrow \{(g_1, \ldots, g_m) \in A'_i \mid g_1 = \ldots = g_{c-1} = 0 \text{ and } g_c \neq 0\}$.
     b) If $S = \emptyset$: `continue`.
     c) Let $s = (s_1, \ldots, s_m) \in S$ be an element with s_c minimal (among the elements in S).

d) Set $\mathfrak{C}[c] \leftarrow s$.

e) For all elements $h = (h_1, \ldots, h_m)$ in the blocks $A'_{i+1}, \ldots, A'_r$:

– Replace h by $h - \left\lfloor \frac{h_c}{s_c} \right\rfloor \cdot s$.

Testing whether $x + N = y + N$ holds is now possible using the following reduction algorithm:

1. Let $t = (t_1, \ldots, t_m) \leftarrow x - y$.
2. For $c \leftarrow 1$ to m:
   a) If $t_c = 0$: `continue`.
   b) Let $s = (s_1, \ldots, s_m) \leftarrow \mathfrak{C}[c]$.
   c) If $s = (0, \ldots, 0)$: `return` "$x + N \neq y + N$".
   d) Set $t \leftarrow t - \left\lfloor \frac{t_c}{s_c} \right\rfloor \cdot s$.
   e) If $t_c \neq 0$: `return` "$x + N \neq y + N$".
3. `Return` "$x + N = y + N$".

Proof. First of all, we have

$$x + N = y + N \;\Leftrightarrow\; x - y + N = N \;\Leftrightarrow\; x - y \in N.$$

Let $d := x - y$.

In this proof, let us call a $(g_1, \ldots, g_m) \in G$ a *c-element*, if $g_1 = \ldots = g_{c-1} = 0$ and $g_c \neq 0$, and define $(0, \ldots, 0)$ to be an $(m+1)$-element.

We now prove in detail why the two algorithms work.

- For all $1 \leq i \leq r$, the block sequences $(A_1, \ldots, A_i)$ and $(A'_1, \ldots, A'_i)$ generate exactly the same elements at any time while the algorithm is running (including the end).

  In order to see why this is true, observe that the innermost replacement operation does not change the set of generated elements, because s is an element of the subgroup $A'_1 + \ldots + A'_i$ (and thus $-\left\lfloor \frac{h_c}{s_c} \right\rfloor \cdot s$ also is an element of this subgroup), and $A'_1 + \ldots + A'_i \leq A'_1 + \ldots + A'_q$ when h is an element in block A'_q (note that $q \geq i + 1$).

  In other words, replacing h by $h - \left\lfloor \frac{h_c}{s_c} \right\rfloor \cdot s$ is a selective shift (Section 5.1.9) using $-\left\lfloor \frac{h_c}{s_c} \right\rfloor \cdot s$ from the subgroup $A'_1 + \ldots + A'_i$ onto h in A'_q (with $q \geq i+1$), and selective shifts never change the set of generated elements.

- The $\mathfrak{C}$ computation algorithm replaces h by $h - \left\lfloor \frac{h_c}{s_c} \right\rfloor \cdot s$. After this step, we clearly have $h_c < s_c$ (from the original h_c, the $-\left\lfloor \frac{h_c}{s_c} \right\rfloor \cdot s$ operation subtracts as many s_c from h_c as possible, thus for the new h_c we get $0 \leq h_c < s_c$).

- $\mathfrak{C}$ has the following properties (after its computation algorithm has finished):
  - $\langle \mathfrak{C} \rangle = N$.
  - For all $1 \leq c \leq m$, the cth element of $\mathfrak{C}$ is a c-element.

  Let $\gamma_c : G \to \mathbb{Z}_{p_c^{k_c}} : (g_1, \ldots, g_m) \mapsto g_c$ the projection of the cth component. Furthermore, let $S(c,i) := \{a \in A'_i \mid a \text{ is a } c\text{-element}\} \cup \{(0,\ldots,0)\}$, with $A'_1, \ldots, A'_r$ being the blocks when entering the ith iteration within the cth outer iteration of the $\mathfrak{C}$ computation algorithm. The following statements hold for all $1 \leq c \leq m$, $1 \leq i \leq r$:
  - $\gamma_c(S(c,1)) \leq \gamma_c(S(c,1) + S(c,2)) \leq \ldots \leq \gamma_c(S(c,1) + \ldots + S(c,r)) \leq \mathbb{Z}_{p_c^{k_c}}$.
  - $\forall\, a \in (S(c,1) + \ldots + S(c,i-1)) \setminus \{(0,\ldots,0)\},\ b \in S(c,i) + \ldots + S(c,r) : \gamma_c(a) > \gamma_c(b)$.

  We prove these properties by induction over c. Let the claims be true for the first $c-1$ components, and let $A'_1, \ldots, A'_r$ be the blocks when entering the cth outer iteration.

  Let i be the first index that the algorithm finds for which $S \neq \emptyset$ (i.e. $S(c,j) = \{(0,\ldots,0)\}$ for all $1 \leq j < i$ and $|S(c,i)| \geq 2$). Let $s = (s_1, \ldots, s_m) \in S$ be an element with s_c minimal, and $o_s := \mathrm{ord}(\gamma_c(s) \text{ in } \mathbb{Z}_{p_c^{k_c}})$. As $A'_1 + \ldots + A'_i \leq G$, all multiples of s (in G) must be contained in $A'_1 + \ldots + A'_i$.

  Let $a_1 \in A'_1$ a c_1-element, …, $a_{i-1} \in A'_{i-1}$ a c_{i-1} element, and $\widetilde{c} := \min\{c_1, \ldots, c_{i-1}\}$. If $\widetilde{c} < c$, from the claims being true for all components $< c$ we obtain that $a_1 + \ldots + a_{i-1}$ is a $\widetilde{c}$-element. Together with the fact that $A'_1, \ldots, A'_{i-1}$ do not contain any c-elements (A'_i is the *first* block containing c-elements), the subgroup $A'_1 + \ldots + A'_{i-1}$ does not contain any c-elements. Keeping in mind that the components $< c$ of the elements in A'_i have already been reduced as much as possible, we can conclude that for every $0 \leq j < o_s$ there must exist an $a \in S$ with $\gamma_c(a) = j \cdot \gamma_c(s)$. This establishes $\gamma_c(S(c,1)) \leq \gamma_c(S(c,1) + S(c,2)) \leq \ldots \leq \gamma_c(S(c,i)) \leq \mathbb{Z}_{p_c^{k_c}}$.

  The full chain $\gamma_c(S(c,1)) \leq \gamma_c(S(c,1) + S(c,2)) \leq \ldots \leq \gamma_c(S(c,1) + \ldots + S(c,r)) \leq \mathbb{Z}_{p_c^{k_c}}$ can be established by iteratively moving into factor groups (imaginarily, only for the proof). First move into the factor group $G/\langle s \rangle$. By this, the c-elements in A'_i become z-elements with $z > c$. Due to the previous reductions, a factor group element in the blocks $A'_{i+1}, \ldots, A'_r$ now is a c-element if and only if it is a c-element in G. The argumentation above can now be applied in $G/\langle s \rangle$ (the first block that contains c-elements will have an index $> i$). Continuing this iteratively until the end of the block sequence, the full chain is established.

  The inner loop of the $\mathfrak{C}$ computation algorithm ensures that the c-component of every element in $A'_{i+1}, \ldots, A'_r$ is smaller than $\gamma_c(s)$, and this holds iteratively for all blocks containing c-elements. Together with the above subgroup chain for $\mathbb{Z}_{p_c^{k_c}}$, we realize that indeed $\forall\, a \in (S(c,1) + \ldots + S(c,i-1)) \setminus \{(0,\ldots,0)\},\ b \in S(c,i) + \ldots + S(c,r) : \gamma_c(a) > \gamma_c(b)$ holds.

- The reduction algorithm tries to reduce d to $(0,\ldots,0)$. If this is possible, $d \in N$ (i.e. $x+N = y+N$), otherwise $d \notin N$ (i.e. $x+N \neq y+N$).

  A reduction is a selective shift of a multiple of an element in $\mathfrak{C}$ onto the current t (which is the element that we want to reduce; it is initially set to d). Each element in $\mathfrak{C}$ originates from a block A'_i and thus is an element of the subgroup N (and all multiples are also elements of N).

  These reductions are performed iteratively for each component. In the first iteration, the algorithm tries to reduce the first component of t. In the second iteration, the second component is reduced, and so on. This process has the property that once the first c components have been reduced, the reduction of the components $c+1, c+2, \ldots$ does not have any effect on the first c components, because in the c'th iteration only a c'-element is used for reduction.

- Recall that each element in $\mathfrak{C}$ originates from a block A'_i. In the reduction algorithm it can happen that t is reduced multiple times by elements originating from a common block. For example, in the jth iteration the jth component of t could be reduced using a j-element u in a block A'_i and later the kth component ($k > j$) of t could be reduced using a k-element v in the same block A'_i. This is not a problem, because $(t-u)-v = t-(u+v)$, and $u+v \in A_1+\ldots+A_i$, as $A_1+\ldots+A_i$ is a subgroup of G. Note that we never need to actually compute the factorization of $u+v$ in $A_1+\ldots+A_i$. It is just important to understand that two such reductions are always equivalent to subtracting *one* element from $A_1+\ldots+A_i$.

  By applying this argument iteratively to all performed reductions, we see that these reductions are equivalent to subtracting one element $w \in N$ from d. Then at the end $t = (0,\ldots,0) \Leftrightarrow \exists\, w \in N : d-w = (0,\ldots,0) \Leftrightarrow d \in N$. □

Proposition 8.14. *Let G be an abelian group (represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$) and $N := A_1 + A_2 + \ldots + A_r$ with $A_1 \leq G$, $A_1+A_2 \leq G$, $A_1+A_2+A_3 \leq G$, …, $N \leq G$. Let $\alpha := (A_1,\ldots,A_r)$.*

Then Algorithm 8.13 performs $O(m \cdot r \cdot \ell(\alpha))$ group operations (and we have $O(m \cdot r \cdot \ell(\alpha)) \subseteq O(\log_2 |G| \cdot \ell(\alpha)^2)$).

Proof. We determine the run-times of the two algorithms:

- $\mathfrak{C}$ computation. Initializing α' involves copying $\ell(\alpha)$ group elements. Initializing $\mathfrak{C}$ requires copying the identity element m times. The outer loop runs exactly m times, and the loop within it runs exactly r times. It is not necessary to actually build the set S; trying to locate a c-element $s = (s_1,\ldots,s_m)$ with s_c minimal is sufficient; so we need to inspect $|A'_i|$ group elements; observe that $|S| \leq |A'_i| = |A_i|$. We consider copying s to $\mathfrak{C}[c]$ to take 1 group operation. The next loop iterates over all elements in the blocks right of A'_i.

  In total, we need

$$O(\ell(\alpha) + m + \sum_{c=1}^{m} \sum_{i=1}^{r} (|A'_i| + 1 + \sum_{j=i+1}^{r} |A'_j|))$$

$$
\begin{aligned}
&= O(\ell(\alpha) + m + \sum_{c=1}^{m}\sum_{i=1}^{r}(|A_i| + 1 + \sum_{j=i+1}^{r}|A_j|)) \\
&\subseteq O(\ell(\alpha) + m + \sum_{c=1}^{m}\sum_{i=1}^{r}(|A_i| + 1 + \ell(\alpha))) \\
&\subseteq O(\ell(\alpha) + m + \sum_{c=1}^{m}\sum_{i=1}^{r}(\ell(\alpha) + \ell(\alpha)) + m \cdot r) \\
&= O(\ell(\alpha) + m + 2 \cdot m \cdot r \cdot \ell(\alpha) + m \cdot r) \\
&= O(m \cdot r \cdot \ell(\alpha))
\end{aligned}
$$

group operations.

- Reduction. We need

$$O(1 + \sum_{c=1}^{m}(1 + 1 + 1)) = O(m)$$

group operations.

Thus, for the $\mathfrak{C}$ computation and the reduction together, we need

$$O(m \cdot r \cdot \ell(\alpha)) + O(m) = O(m \cdot r \cdot \ell(\alpha))$$

group operations. □

Lemma 8.15. *Let G be an abelian group (represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$) and $N := A_1 + A_2 + \ldots + A_r$ with $A_1 \leq G$, $A_1 + A_2 \leq G$, $A_1 + A_2 + A_3 \leq G$, …, $N \leq G$. Let $\alpha := (A_1, \ldots, A_r)$ and $S \subseteq G/N$ (each element of S shall be given by a representative in G).*

Then testing whether S is a subgroup of G/N is possible using $O(|S|^3 \cdot m \cdot r \cdot \ell(\alpha))$ group operations.

The algorithm for testing this takes α and S as input. The set N (containing $|A_1| \cdot |A_2| \cdots |A_r|$ elements) is not *part of the input; it is only used for notation.*

Proof. S is a subgroup, if $a - b \in S$ for all $a, b \in S$. So, compute all possible differences $d := a - b$ (there are $|S|^2$ such differences) and test whether $d \in S$ holds (by searching an $s \in S$ with $d + N = s + N$; this can be tested using Algorithm 8.13). For this, we need

$$|S|^2 + |S|^2 \cdot |S| \cdot O(m \cdot r \cdot \ell(\alpha)) = O(|S|^3 \cdot m \cdot r \cdot \ell(\alpha))$$

group operations. □

Algorithm 8.16. Let G be an abelian group (represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$), $\beta = (B_1, B_2, \ldots, B_n) \in \Lambda(G)$ with $|B_i| \in \mathbb{P}$ for all

$1 \leq i \leq n$, and $h \in G$. Additionally, we can assume that β is canonical (see Proposition 7.1).

The following recursive algorithm finds the factorization of h with respect to β, when calling `FactorizeAb`$(\beta, h, ((0, \ldots, 0)))$:

Function `FactorizeAb`(LogSig α, GroupElement g, BlockSequence N) : List<UInt>

1. If $|\alpha| = 1$: find $g + N$ in $\alpha[1]$ (in G/N) and `return` its index (one item in a list).
2. Let s be the index of the first block in α being a subgroup in G/N.
3. Let $\alpha' \leftarrow \alpha$ without the sth block. Let $N' \leftarrow N$ with block $\alpha[s]$ appended.
4. Let $v \leftarrow$ `FactorizeAb`(α', g, N').
5. Insert 0 at index s in v.
6. Let $t \leftarrow \alpha[1][v[1]] + \ldots + \alpha[s-1][v[s-1]] + \alpha[s+1][v[s+1]] + \ldots + \alpha[|\alpha|][v[|\alpha|]]$.
7. For $i \leftarrow 1$ to $|\alpha[s]|$:
   - If $t + \alpha[s][i] + N = g + N$: set $v[s] \leftarrow i$ and `return` v.

Proof. The algorithm realizes the idea of Theorem 7.6.

N is fulfilling the requirements for Algorithm 8.13 and Lemma 8.15: initially we have $N = \{(0, \ldots, 0)\}$ (which indeed is a subgroup of G) and on every recursion level a block $\alpha[s]$ is appended, where $\alpha[s]$ is a subgroup in G/N (such a block always exists by Rédei's theorem); i.e. every new N' is also a subgroup of G and while descending in the recursion we get sequences of subgroups as required by Algorithm 8.13 and Lemma 8.15. □

Proposition 8.17. *Let G be an abelian group (represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$), $\beta = (B_1, B_2, \ldots, B_n) \in \Lambda(G)$ canonical with $|B_i| \in \mathbb{P}$ for all $1 \leq i \leq n$. Let $q := \max\{|B_i| \mid 1 \leq i \leq n\}$.*

Then Algorithm 8.16 requires $O(q^3 \cdot m \cdot n^3 \cdot \ell(\beta))$ group operations.

Proof. We count the required group operations:

- Searching for $g + N$ in a block (if α consists of one block only) requires at most q equality tests using Algorithm 8.13, i.e. $q \cdot O(m \cdot n \cdot \ell(\beta))$ group operations.
- For finding the first block being a subgroup in G/N we use Lemma 8.15 for each block and thus need at most $|\alpha| \cdot O(q^3 \cdot m \cdot n \cdot \ell(\beta))$ group operations.
- Computing t and completing the factorization requires at most

$$(n-2) + |\alpha[s]| \cdot (1 + O(m \cdot n \cdot \ell(\beta))) \subseteq O(q \cdot m \cdot n \cdot \ell(\beta))$$

  group operations.

In total, we need

$$\begin{aligned}
&\underbrace{q \cdot O(m \cdot n \cdot \ell(\beta))}_{\text{Case 1 block}} + \sum_{i=2}^{n} \left(\underbrace{i \cdot O(q^3 \cdot m \cdot n \cdot \ell(\beta))}_{\text{Subgroup tests}} + \underbrace{O(q \cdot m \cdot n \cdot \ell(\beta))}_{\text{Completing fac.}} \right) \\
&= n \cdot O(q \cdot m \cdot n \cdot \ell(\beta)) + O(q^3 \cdot m \cdot n \cdot \ell(\beta)) \cdot \sum_{i=2}^{n} i \\
&= O(q^3 \cdot m \cdot n^3 \cdot \ell(\beta))
\end{aligned}$$

group operations. □

Implementation. This algorithm, together with some extensions, has been implemented in our LogSig program. For the command line syntax, see Section 12.3.

Example 8.18. Let $G = \mathbb{Z}_9 \oplus \mathbb{Z}_9 \oplus \mathbb{Z}_3 \oplus \mathbb{Z}_2$, $g = (4, 3, 0, 1)$, and

$$\alpha = \begin{pmatrix} (0,0,0,0) \\ (0,6,2,0) \\ (3,0,2,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (3,6,1,0) \\ (6,3,2,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (0,8,1,0) \\ (3,1,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (3,0,0,0) \\ (3,0,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (3,6,0,1) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (2,3,1,1) \\ (7,0,1,0) \end{pmatrix}$$

a logarithmic signature for G. We are looking for a factorization of g with respect to α.

$$\alpha_0 := \begin{pmatrix} (0,0,0,0) \\ (0,6,2,0) \\ (3,0,2,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (3,6,1,0) \\ (6,3,2,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (0,8,1,0) \\ (3,1,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (3,0,0,0) \\ (3,0,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (3,6,0,1) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (2,3,1,1) \\ (7,0,1,0) \end{pmatrix},$$
$$N_0 := \{(0,0,0,0)\},$$
$$N_0' := \{(0,0,0,0)\}.$$

The second block in α_0 is a subgroup (in G/N_0).

$$\alpha_1 := \begin{pmatrix} (0,0,0,0) \\ (0,6,2,0) \\ (3,0,2,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (0,8,1,0) \\ (3,1,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (3,0,0,0) \\ (3,0,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (3,6,0,1) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (2,3,1,1) \\ (7,0,1,0) \end{pmatrix},$$
$$N_1 := \begin{pmatrix} (0,0,0,0) \\ (3,6,1,0) \\ (6,3,2,0) \end{pmatrix},$$
$$N_1' := \begin{pmatrix} (0,0,0,0) \\ (3,6,1,0) \\ (6,3,2,0) \end{pmatrix}.$$

The first block in α_1 is a subgroup (in G/N_1).

$$\alpha_2 := \begin{pmatrix} (0,0,0,0) \\ (0,8,1,0) \\ (3,1,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (3,0,0,0) \\ (3,0,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (3,6,0,1) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (2,3,1,1) \\ (7,0,1,0) \end{pmatrix},$$
$$N_2 := \begin{pmatrix} (0,0,0,0) \\ (3,6,1,0) \\ (6,3,2,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (0,6,2,0) \\ (3,0,2,0) \end{pmatrix},$$
$$N_2' := \begin{pmatrix} (0,0,0,0) \\ (3,6,1,0) \\ (6,3,2,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (0,3,1,0) \\ (0,6,2,0) \end{pmatrix}.$$

The second block in α_2 is a subgroup (in G/N_2).

$$\begin{aligned}
\alpha_3 &:= \begin{pmatrix}(0,0,0,0)\\(0,8,1,0)\\(3,1,1,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(3,6,0,1)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(2,3,1,1)\\(7,0,1,0)\end{pmatrix},\\
N_3 &:= \begin{pmatrix}(0,0,0,0)\\(3,6,1,0)\\(6,3,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,6,2,0)\\(3,0,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(3,0,0,0)\\(3,0,1,0)\end{pmatrix},\\
N_3' &:= \begin{pmatrix}(0,0,0,0)\\(3,6,1,0)\\(6,3,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,3,1,0)\\(0,6,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,1,0)\\(0,0,2,0)\end{pmatrix}.
\end{aligned}$$

The first block in α_3 is a subgroup (in G/N_3).

$$\begin{aligned}
\alpha_4 &:= \begin{pmatrix}(0,0,0,0)\\(3,6,0,1)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(2,3,1,1)\\(7,0,1,0)\end{pmatrix},\\
N_4 &:= \begin{pmatrix}(0,0,0,0)\\(3,6,1,0)\\(6,3,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,6,2,0)\\(3,0,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(3,0,0,0)\\(3,0,1,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,8,1,0)\\(3,1,1,0)\end{pmatrix},\\
N_4' &:= \begin{pmatrix}(0,0,0,0)\\(3,6,1,0)\\(6,3,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,3,1,0)\\(0,6,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,1,0)\\(0,0,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,1,0,0)\\(0,2,0,0)\end{pmatrix}.
\end{aligned}$$

The first block in α_4 is a subgroup (in G/N_4).

$$\begin{aligned}
\alpha_5 &:= \begin{pmatrix}(0,0,0,0)\\(2,3,1,1)\\(7,0,1,0)\end{pmatrix},\\
N_5 &:= \begin{pmatrix}(0,0,0,0)\\(3,6,1,0)\\(6,3,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,6,2,0)\\(3,0,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(3,0,0,0)\\(3,0,1,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,8,1,0)\\(3,1,1,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(3,6,0,1)\end{pmatrix},\\
N_5' &:= \begin{pmatrix}(0,0,0,0)\\(3,6,1,0)\\(6,3,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,3,1,0)\\(0,6,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,1,0)\\(0,0,2,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,1,0,0)\\(0,2,0,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,0,1)\end{pmatrix}.
\end{aligned}$$

Factorization of $g + N_5$ in α_5:

$$\alpha_5 = \begin{pmatrix}(0,0,0,0)\\(2,3,1,1)\\\boxed{(7,0,1,0)}\end{pmatrix}.$$

Factorization of $g + N_4$ in α_4:

$$\alpha_4 = \begin{pmatrix}(0,0,0,0)\\\boxed{(3,6,0,1)}\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(2,3,1,1)\\\boxed{(7,0,1,0)}\end{pmatrix}.$$

Factorization of $g + N_3$ in α_3:

$$\alpha_3 = \begin{pmatrix}\boxed{(0,0,0,0)}\\(0,8,1,0)\\(3,1,1,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\\boxed{(3,6,0,1)}\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(2,3,1,1)\\\boxed{(7,0,1,0)}\end{pmatrix}.$$

Factorization of $g + N_2$ in α_2:

$$\alpha_2 = \begin{pmatrix} \boxed{(0,0,0,0)} \\ (0,8,1,0) \\ (3,1,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ \boxed{(3,0,0,0)} \\ (3,0,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ \boxed{(3,6,0,1)} \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (2,3,1,1) \\ \boxed{(7,0,1,0)} \end{pmatrix}.$$

Factorization of $g + N_1$ in α_1:

$$\alpha_1 = \begin{pmatrix} (0,0,0,0) \\ \boxed{(0,6,2,0)} \\ (3,0,2,0) \end{pmatrix} \begin{pmatrix} \boxed{(0,0,0,0)} \\ (0,8,1,0) \\ (3,1,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ \boxed{(3,0,0,0)} \\ (3,0,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ \boxed{(3,6,0,1)} \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (2,3,1,1) \\ \boxed{(7,0,1,0)} \end{pmatrix}.$$

Factorization of $g + N_0$ in α_0:

$$\alpha_0 = \begin{pmatrix} (0,0,0,0) \\ \boxed{(0,6,2,0)} \\ (3,0,2,0) \end{pmatrix} \begin{pmatrix} \boxed{(0,0,0,0)} \\ (3,6,1,0) \\ (6,3,2,0) \end{pmatrix} \begin{pmatrix} \boxed{(0,0,0,0)} \\ (0,8,1,0) \\ (3,1,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ \boxed{(3,0,0,0)} \\ (3,0,1,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ \boxed{(3,6,0,1)} \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (2,3,1,1) \\ \boxed{(7,0,1,0)} \end{pmatrix}.$$

Example 8.19. Let $G = \mathbb{Z}_3 \oplus \mathbb{Z}_3 \oplus \mathbb{Z}_9 \oplus \mathbb{Z}_8$, $g = (1,1,7,7)$, and

$$\alpha = \begin{pmatrix} (0,0,0,0) \\ (1,0,0,7) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,6,0) \\ (0,0,6,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (2,1,6,4) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,0,6,0) \\ (2,0,3,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,3,0) \\ (2,1,6,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (0,2,5,4) \\ (1,2,4,2) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,6,6) \end{pmatrix}$$

a logarithmic signature for G. We are looking for a factorization of g with respect to α.

$$\alpha_0 := \begin{pmatrix} (0,0,0,0) \\ (1,0,0,7) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,6,0) \\ (0,0,6,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (2,1,6,4) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,0,6,0) \\ (2,0,3,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,3,0) \\ (2,1,6,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (0,2,5,4) \\ (1,2,4,2) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,6,6) \end{pmatrix},$$
$$N_0 := \{(0,0,0,0)\},$$
$$N_0' := \{(0,0,0,0)\}.$$

The 4th block in α_0 is a subgroup (in G/N_0).

$$\alpha_1 := \begin{pmatrix} (0,0,0,0) \\ (1,0,0,7) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,6,0) \\ (0,0,6,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (2,1,6,4) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,3,0) \\ (2,1,6,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (0,2,5,4) \\ (1,2,4,2) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,6,6) \end{pmatrix},$$
$$N_1 := \begin{pmatrix} (0,0,0,0) \\ (1,0,6,0) \\ (2,0,3,0) \end{pmatrix},$$
$$N_1' := \begin{pmatrix} (0,0,0,0) \\ (1,0,6,0) \\ (2,0,3,0) \end{pmatrix}.$$

The 4th block in α_1 is a subgroup (in G/N_1).

$$\alpha_2 := \begin{pmatrix} (0,0,0,0) \\ (1,0,0,7) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,6,0) \\ (0,0,6,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (2,1,6,4) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (0,2,5,4) \\ (1,2,4,2) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,6,6) \end{pmatrix},$$
$$N_2 := \begin{pmatrix} (0,0,0,0) \\ (1,0,6,0) \\ (2,0,3,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,2,3,0) \\ (2,1,6,0) \end{pmatrix},$$
$$N_2' := \begin{pmatrix} (0,0,0,0) \\ (1,0,6,0) \\ (2,0,3,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (0,1,3,0) \\ (0,2,6,0) \end{pmatrix}.$$

The second block in α_2 is a subgroup (in G/N_2).

$$\alpha_3 := \begin{pmatrix}(0,0,0,0)\\(1,0,0,7)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(2,1,6,4)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,2,5,4)\\(1,2,4,2)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,6,6)\end{pmatrix},$$
$$N_3 := \begin{pmatrix}(0,0,0,0)\\(1,0,6,0)\\(2,0,3,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,3,0)\\(2,1,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,6,0)\\(0,0,6,0)\end{pmatrix},$$
$$N_3' := \begin{pmatrix}(0,0,0,0)\\(1,0,6,0)\\(2,0,3,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,1,3,0)\\(0,2,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,3,0)\\(0,0,6,0)\end{pmatrix}.$$

The second block in α_3 is a subgroup (in G/N_3).

$$\alpha_4 := \begin{pmatrix}(0,0,0,0)\\(1,0,0,7)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,2,5,4)\\(1,2,4,2)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,6,6)\end{pmatrix},$$
$$N_4 := \begin{pmatrix}(0,0,0,0)\\(1,0,6,0)\\(2,0,3,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,3,0)\\(2,1,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,6,0)\\(0,0,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(2,1,6,4)\end{pmatrix},$$
$$N_4' := \begin{pmatrix}(0,0,0,0)\\(1,0,6,0)\\(2,0,3,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,1,3,0)\\(0,2,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,3,0)\\(0,0,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,0,4)\end{pmatrix}.$$

The third block in α_4 is a subgroup (in G/N_4).

$$\alpha_5 := \begin{pmatrix}(0,0,0,0)\\(1,0,0,7)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,2,5,4)\\(1,2,4,2)\end{pmatrix},$$
$$N_5 := \begin{pmatrix}(0,0,0,0)\\(1,0,6,0)\\(2,0,3,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,3,0)\\(2,1,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,6,0)\\(0,0,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(2,1,6,4)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,6,6)\end{pmatrix},$$
$$N_5' := \begin{pmatrix}(0,0,0,0)\\(1,0,6,0)\\(2,0,3,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,1,3,0)\\(0,2,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,3,0)\\(0,0,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,0,4)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,0,2)\end{pmatrix}.$$

The first block in α_5 is a subgroup (in G/N_5).

$$\alpha_6 := \begin{pmatrix}(0,0,0,0)\\(0,2,5,4)\\(1,2,4,2)\end{pmatrix},$$
$$N_6 := \begin{pmatrix}(0,0,0,0)\\(1,0,6,0)\\(2,0,3,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,3,0)\\(2,1,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,6,0)\\(0,0,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(2,1,6,4)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,2,6,6)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(1,0,0,7)\end{pmatrix},$$
$$N_6' := \begin{pmatrix}(0,0,0,0)\\(1,0,6,0)\\(2,0,3,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,1,3,0)\\(0,2,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,3,0)\\(0,0,6,0)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,0,4)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,0,2)\end{pmatrix}\begin{pmatrix}(0,0,0,0)\\(0,0,0,1)\end{pmatrix}.$$

Factorization of $g + N_6$ in α_6:

$$\alpha_6 = \begin{pmatrix}(0,0,0,0)\\(0,2,5,4)\\\boxed{(1,2,4,2)}\end{pmatrix}.$$

Factorization of $g + N_5$ in α_5:

$$\alpha_5 = \left(\boxed{(0,0,0,0)}\right)\left(\begin{matrix}(0,0,0,0)\\(0,2,5,4)\\\boxed{(1,2,4,2)}\end{matrix}\right).$$

Factorization of $g + N_4$ in α_4:

$$\alpha_4 = \left(\boxed{(0,0,0,0)}\right)\left(\begin{matrix}(0,0,0,0)\\(0,2,5,4)\\\boxed{(1,2,4,2)}\end{matrix}\right)\left(\boxed{(0,0,0,0)}\right).$$

Factorization of $g + N_3$ in α_3:

$$\alpha_3 = \left(\boxed{(0,0,0,0)}\right)\left(\boxed{(0,0,0,0)}\right)\left(\begin{matrix}(0,0,0,0)\\(0,2,5,4)\\\boxed{(1,2,4,2)}\end{matrix}\right)\left(\boxed{(0,0,0,0)}\right).$$

Factorization of $g + N_2$ in α_2:

$$\alpha_2 = \left(\boxed{(0,0,0,0)}\right)\left(\begin{matrix}\boxed{(0,0,0,0)}\\(1,2,6,0)\\(0,0,6,0)\end{matrix}\right)\left(\boxed{(0,0,0,0)}\right)\left(\begin{matrix}(0,0,0,0)\\(0,2,5,4)\\\boxed{(1,2,4,2)}\end{matrix}\right)\left(\boxed{(0,0,0,0)}\right).$$

Factorization of $g + N_1$ in α_1:

$$\alpha_1 = \left(\boxed{(0,0,0,0)}\right)\left(\begin{matrix}\boxed{(0,0,0,0)}\\(1,2,6,0)\\(0,0,6,0)\end{matrix}\right)\left(\boxed{(0,0,0,0)}\right)\left(\begin{matrix}\boxed{(0,0,0,0)}\\(1,2,3,0)\\(2,1,6,0)\end{matrix}\right)\left(\begin{matrix}(0,0,0,0)\\(0,2,5,4)\\\boxed{(1,2,4,2)}\end{matrix}\right)\left(\boxed{(0,0,0,0)}\right).$$

Factorization of $g + N_0$ in α_0: $\alpha_0 =$

$$\left(\boxed{(0,0,0,0)}\right)\left(\begin{matrix}\boxed{(0,0,0,0)}\\(1,2,6,0)\\(0,0,6,0)\end{matrix}\right)\left(\boxed{(0,0,0,0)}\right)\left(\begin{matrix}(0,0,0,0)\\\boxed{(1,0,6,0)}\\(2,0,3,0)\end{matrix}\right)\left(\begin{matrix}\boxed{(0,0,0,0)}\\(1,2,3,0)\\(2,1,6,0)\end{matrix}\right)\left(\begin{matrix}(0,0,0,0)\\(0,2,5,4)\\\boxed{(1,2,4,2)}\end{matrix}\right)\left(\boxed{(0,0,0,0)}\right).$$

8.5. Factoring in Logarithmic Signatures with Blocks of Composite Size

Observation 8.20. If a logarithmic signature α of some abelian group G contains a subgroup block A_i of composite size, Algorithm 8.16 is able to compute factorizations in G, if it can compute factorizations in G/A_i. One can easily check that none of the simplifications/computations in the algorithm actually requires the blocks to be of prime size; this was only required to prove that in arbitrary abelian groups (represented as direct sum of cyclic groups) logarithmic signatures with blocks of prime size are tame.

So, the algorithm can successfully factor elements, if on every recursion level it encounters a subgroup block, independent of the size of the block.

Example 8.21. Let $G = \mathbb{Z}_8 \oplus \mathbb{Z}_4$, $g = (5, 3)$, and

$$\alpha = \begin{pmatrix}(0,0)\\(2,1)\end{pmatrix}\begin{pmatrix}(0,0)\\(2,0)\\(4,0)\\(6,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,2)\end{pmatrix}\begin{pmatrix}(0,0)\\(4,2)\end{pmatrix}$$

a logarithmic signature for G. We are looking for a factorization of g with respect to α.

$$\alpha_0 := \begin{pmatrix}(0,0)\\(2,1)\end{pmatrix}\begin{pmatrix}(0,0)\\(2,0)\\(4,0)\\(6,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,2)\end{pmatrix}\begin{pmatrix}(0,0)\\(4,2)\end{pmatrix},$$
$$N_0 := \{(0,0)\},$$
$$N_0' := \{(0,0)\}.$$

The second block in α_0 is a subgroup (in G/N_0).

$$\alpha_1 := \begin{pmatrix}(0,0)\\(2,1)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,2)\end{pmatrix}\begin{pmatrix}(0,0)\\(4,2)\end{pmatrix},$$
$$N_1 := \begin{pmatrix}(0,0)\\(2,0)\\(4,0)\\(6,0)\end{pmatrix},$$
$$N_1' := \begin{pmatrix}(0,0)\\(2,0)\\(4,0)\\(6,0)\end{pmatrix}.$$

The second block in α_1 is a subgroup (in G/N_1).

$$\alpha_2 := \begin{pmatrix}(0,0)\\(2,1)\end{pmatrix}\begin{pmatrix}(0,0)\\(4,2)\end{pmatrix},$$
$$N_2 := \begin{pmatrix}(0,0)\\(2,0)\\(4,0)\\(6,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,2)\end{pmatrix},$$
$$N_2' := \begin{pmatrix}(0,0)\\(2,0)\\(4,0)\\(6,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,2)\end{pmatrix}.$$

The second block in α_2 is a subgroup (in G/N_2).

$$\alpha_3 := \begin{pmatrix}(0,0)\\(2,1)\end{pmatrix},$$
$$N_3 := \begin{pmatrix}(0,0)\\(2,0)\\(4,0)\\(6,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,2)\end{pmatrix}\begin{pmatrix}(0,0)\\(4,2)\end{pmatrix},$$
$$N_3' := \begin{pmatrix}(0,0)\\(2,0)\\(4,0)\\(6,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,2)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,2)\end{pmatrix}.$$

Factorization of $g + N_3$ in α_3:

$$\alpha_3 = \left(\begin{matrix}(0,0)\\ \boxed{(2,1)}\end{matrix}\right).$$

Factorization of $g + N_2$ in α_2:

$$\alpha_2 = \left(\begin{matrix}(0,0)\\ \boxed{(2,1)}\end{matrix}\right)\left(\begin{matrix}\boxed{(0,0)}\\ (4,2)\end{matrix}\right).$$

Factorization of $g + N_1$ in α_1:

$$\alpha_1 = \left(\begin{matrix}(0,0)\\ \boxed{(2,1)}\end{matrix}\right)\left(\begin{matrix}(0,0)\\ \boxed{(1,2)}\end{matrix}\right)\left(\begin{matrix}\boxed{(0,0)}\\ (4,2)\end{matrix}\right).$$

Factorization of $g + N_0$ in α_0:

$$\alpha_0 = \left(\begin{matrix}(0,0)\\ \boxed{(2,1)}\end{matrix}\right)\left(\begin{matrix}(0,0)\\ \boxed{(2,0)}\\ (4,0)\\ (6,0)\end{matrix}\right)\left(\begin{matrix}(0,0)\\ \boxed{(1,2)}\end{matrix}\right)\left(\begin{matrix}\boxed{(0,0)}\\ (4,2)\end{matrix}\right).$$

Logarithmic signatures containing composite-sized blocks do not necessarily contain subgroup blocks. However, we can try to decompose composite-sized blocks, in the hope that one of the smaller blocks is a subgroup (such that we can descend one recursion level in the algorithm and work with a simpler logarithmic signature). In the following we present such a decomposition.

Lemma 8.22. *Let G be an abelian group (represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$) and $N := A_1 + A_2 + \ldots + A_r$ with $A_1 \leq G$, $A_1 + A_2 \leq G$, $A_1 + A_2 + A_3 \leq G$, …, $N \leq G$. Let $\alpha := (A_1, \ldots, A_r)$ and $S \subseteq G/N$ with $(0, \ldots, 0) \in S$ (each element of S shall be given by a representative in G).*

Then finding the periods of S (representatives in G for elements in G/N) is possible using $O(|S|^3 \cdot m \cdot r \cdot \ell(\alpha))$ group operations.

The algorithm for computing the periods takes α and S as input. The set N (containing $|A_1| \cdot |A_2| \cdots |A_r|$ elements) is not *part of the input; it is only used for notation.*

Proof. A $g \in G/N \setminus \{(0, \ldots, 0)\}$ is a period of S, if $S + g = S$. As $(0, \ldots, 0) \in S$, the only candidates for g are the elements in S.

So, for each candidate $g \in S \setminus \{(0, \ldots, 0)\}$ we build the set $S + g$ and test whether $S + g = S$, which can be done using at most $|S|^2$ element equality tests using Algorithm 8.13 (for each element in $S + g$ check whether it also is in S).

In total, we need

$$|S| \cdot \big(|S| + |S|^2 \cdot O(m \cdot r \cdot \ell(\alpha))\big) = O(|S|^3 \cdot m \cdot r \cdot \ell(\alpha))$$

group operations. □

Algorithm 8.23. We extend Algorithm 8.16 in the following way to additionally support factoring periodic blocks of composite size.

Let the input be the same as in Algorithm 8.16.

If no subgroup can be found on the current recursion level, locate all composite-sized blocks and test them for periodicity. If a block A_i is of composite size and periodic, by Lemma 2.7 it can be decomposed as $A_i = A + S$ with $A \subseteq G/N$ and $S \leq G/N$. Replace A_i by (A, S) (two blocks) and run the algorithm on this new logarithmic signature. If we find a factorization in the new logarithmic signature, deriving a factorization in the original one is easy (just search the correct element in A_i).

Proposition 8.24. *Algorithm 8.23 requires $O(\log_2 |G| \cdot q^3 \cdot m \cdot n^2 \cdot \ell(\beta))$ group operations.*

Proof. Finding the periods of a block is possible using at most $O(q^3 \cdot m \cdot n \cdot \ell(\beta))$ group operations by Lemma 8.22.

The block A can then be constructed as follows: start with $A = \emptyset$, pick an arbitrary a in A_i, put a into A and remove the set $a + S$ from A_i, and repeat this until $A_i = \emptyset$. This procedure requires at most

$$\frac{|A_i|}{|S|} \cdot (|S| + |S| \cdot |A_i| \cdot O(m \cdot n \cdot \ell(\beta))) \subseteq O(q^2 \cdot m \cdot n \cdot \ell(\beta))$$

group operations.

Splitting a periodic block can happen at most $\sum_{i=1}^{m} k_i - 1 < \log_2 |G|$ times. Each time a periodic block is splitted, another recursion level is entered (for a logarithmic signature having at most $n + 1$ blocks), where at most $n + 1$ subgroup block tests are performed (requiring at most $(n+1) \cdot O(q^3 \cdot m \cdot n \cdot \ell(\beta))$ group operations) until finding the subgroup block from the splitting.

In total, we need

$$\begin{aligned}
&\underbrace{q \cdot O(m \cdot n \cdot \ell(\beta))}_{\text{Case 1 block}} + \sum_{i=2}^{n} \left(\underbrace{i \cdot 2 \cdot O(q^3 \cdot m \cdot n \cdot \ell(\beta))}_{\text{Subgroup tests and finding periods}} + \underbrace{O(q \cdot m \cdot n \cdot \ell(\beta))}_{\text{Completing fac.}} \right) + \\
&(\sum_{i=1}^{m} k_i - 1) \cdot \left(\underbrace{O(q^2 \cdot m \cdot n \cdot \ell(\beta))}_{\text{Split to } A+S} + \underbrace{(n+1) \cdot O(q^3 \cdot m \cdot n \cdot \ell(\beta))}_{\text{Subgroup tests on new rec. level}} + \underbrace{O(q \cdot m \cdot n \cdot \ell(\beta))}_{\text{Completing fac.}} \right) \\
&= O(q^3 \cdot m \cdot n^3 \cdot \ell(\beta)) + \log_2 |G| \cdot O(q^3 \cdot m \cdot n^2 \cdot \ell(\beta)) \\
&= O(\log_2 |G| \cdot q^3 \cdot m \cdot n^2 \cdot \ell(\beta))
\end{aligned}$$

group operations. □

Example 8.25. Let $G = \mathbb{Z}_{16} \oplus \mathbb{Z}_2$, $g = (7, 0)$, and

$$\alpha = \begin{pmatrix} (0,0) \\ (2,0) \end{pmatrix} \begin{pmatrix} (0,0) \\ (1,0) \\ (4,0) \\ (5,1) \end{pmatrix} \begin{pmatrix} (0,0) \\ (8,0) \end{pmatrix} \begin{pmatrix} (0,0) \\ (0,1) \end{pmatrix}$$

a logarithmic signature for G. We are looking for a factorization of g with respect to α.

$$\alpha_0 := \begin{pmatrix}(0,0)\\(2,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,0)\\(4,0)\\(5,1)\end{pmatrix}\begin{pmatrix}(0,0)\\(8,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,1)\end{pmatrix},$$
$$N_0 := \{(0,0)\},$$
$$N_0' := \{(0,0)\}.$$

The third block in α_0 is a subgroup (in G/N_0).

$$\alpha_1 := \begin{pmatrix}(0,0)\\(2,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,0)\\(4,0)\\(5,1)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,1)\end{pmatrix},$$
$$N_1 := \begin{pmatrix}(0,0)\\(8,0)\end{pmatrix},$$
$$N_1' := \begin{pmatrix}(0,0)\\(8,0)\end{pmatrix}.$$

The third block in α_1 is a subgroup (in G/N_1).

$$\alpha_2 := \begin{pmatrix}(0,0)\\(2,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,0)\\(4,0)\\(5,1)\end{pmatrix},$$
$$N_2 := \begin{pmatrix}(0,0)\\(8,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,1)\end{pmatrix},$$
$$N_2' := \begin{pmatrix}(0,0)\\(8,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,1)\end{pmatrix}.$$

The second block is periodic (in G/N_2) and can be splitted into two blocks.

$$\alpha_3 := \begin{pmatrix}(0,0)\\(2,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(4,0)\end{pmatrix},$$
$$N_3 := \begin{pmatrix}(0,0)\\(8,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,1)\end{pmatrix},$$
$$N_3' := \begin{pmatrix}(0,0)\\(8,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,1)\end{pmatrix}.$$

The third block in α_3 is a subgroup (in G/N_3).

$$\alpha_4 := \begin{pmatrix}(0,0)\\(2,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(1,0)\end{pmatrix},$$
$$N_4 := \begin{pmatrix}(0,0)\\(8,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,1)\end{pmatrix}\begin{pmatrix}(0,0)\\(4,0)\end{pmatrix},$$
$$N_4' := \begin{pmatrix}(0,0)\\(8,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,1)\end{pmatrix}\begin{pmatrix}(0,0)\\(4,0)\end{pmatrix}.$$

The first block in α_4 is a subgroup (in G/N_4).

$$\alpha_5 := \begin{pmatrix}(0,0)\\(1,0)\end{pmatrix},$$
$$N_5 := \begin{pmatrix}(0,0)\\(8,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,1)\end{pmatrix}\begin{pmatrix}(0,0)\\(4,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(2,0)\end{pmatrix},$$
$$N_5' := \begin{pmatrix}(0,0)\\(8,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,1)\end{pmatrix}\begin{pmatrix}(0,0)\\(4,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(2,0)\end{pmatrix}.$$

Factorization of $g + N_5$ in α_5:

$$\alpha_5 = \left(\begin{array}{c}(0,0)\\ \hline\boxed{(1,0)}\end{array}\right).$$

Factorization of $g + N_4$ in α_4:

$$\alpha_4 = \left(\begin{array}{c}(0,0)\\ \hline\boxed{(2,0)}\end{array}\right)\left(\begin{array}{c}(0,0)\\ \hline\boxed{(1,0)}\end{array}\right).$$

Factorization of $g + N_3$ in α_3:

$$\alpha_3 = \left(\begin{array}{c}(0,0)\\ \hline\boxed{(2,0)}\end{array}\right)\left(\begin{array}{c}(0,0)\\ \hline\boxed{(1,0)}\end{array}\right)\left(\begin{array}{c}(0,0)\\ \hline\boxed{(4,0)}\end{array}\right).$$

Factorization of $g + N_2$ in α_2:

$$\alpha_2 = \left(\begin{array}{c}(0,0)\\ \hline\boxed{(2,0)}\end{array}\right)\left(\begin{array}{c}(0,0)\\ (1,0)\\ (4,0)\\ \hline\boxed{(5,1)}\end{array}\right).$$

Factorization of $g + N_1$ in α_1:

$$\alpha_1 = \left(\begin{array}{c}(0,0)\\ \hline\boxed{(2,0)}\end{array}\right)\left(\begin{array}{c}(0,0)\\ (1,0)\\ (4,0)\\ \hline\boxed{(5,1)}\end{array}\right)\left(\begin{array}{c}(0,0)\\ \hline\boxed{(0,1)}\end{array}\right).$$

Factorization of $g + N_0$ in α_0:

$$\alpha_0 = \left(\begin{array}{c}(0,0)\\ \hline\boxed{(2,0)}\end{array}\right)\left(\begin{array}{c}(0,0)\\ (1,0)\\ (4,0)\\ \hline\boxed{(5,1)}\end{array}\right)\left(\begin{array}{c}\boxed{(0,0)}\\ \hline(8,0)\end{array}\right)\left(\begin{array}{c}(0,0)\\ \hline\boxed{(0,1)}\end{array}\right).$$

Implementation. Algorithm 8.23 has been implemented in our LogSig program. For the command line syntax, see Section 12.3.

8.6. Multiple Factorizations

We now enhance Algorithm 8.23 to support factoring with respect to certain multiple factorizations of abelian groups.

Definition 8.26. Let G be an abelian group and $\alpha = (A_1, \ldots, A_n)$ a k-factorization of G, i.e. every element in G can be expressed in exactly k different ways with respect to α.

We define a function μ that maps α to another k-factorization $\mu(\alpha)$ of G: for every block A_i in α test whether there exists $2 \leq s < |A_i|$, such that every element in A_i occurs exactly s times in A_i, and if so, replace A_i by a block $(0, \ldots, 0)$ of size s and a block A_i' of size $\frac{|A_i|}{s}$, where A_i' contains all elements of A_i once (i.e. without duplicates).

Example 8.27. Let $G = \mathbb{Z}_{2^4}$.

- $\alpha := ((0,1),(0,2),(0,4),(0,8),(0,0))$ is a 2-factorization of G, and $\mu(\alpha) = \alpha$.
- $\alpha := ((0,1),(0,2,0,2),(0,4),(0,8,8,0,0,8))$ is a 6-factorization of G, and
$$\mu(\alpha) = ((0,1),(0,0),(0,2),(0,4),(0,0,0),(0,8)).$$
- $\alpha := ((0,1,2,3),(0,4,8,12),(0,1,2,3))$ is a 4-factorization of G, and $\mu(\alpha) = \alpha$.
- $\alpha := ((0,1,2,3),(0,4,8,12),(0,1,1,2,2,2))$ is a 6-factorization of G, and $\mu(\alpha) = \alpha$.
- $\alpha := ((0,1,2,3),(0,4,8,12),(0,1,1,0))$ is a 4-factorization of G, and
$$\mu(\alpha) = ((0,1,2,3),(0,4,8,12),(0,0),(0,1)).$$

Algorithm 8.28. We enhance Algorithm 8.23 to factor elements in certain multiple factorizations. Let G be an abelian group and $\alpha = (A_1, \ldots, A_n)$ a k-factorization of G (i.e. every element in G can be expressed in exactly k different ways with respect to α).

On every recursion level: if $G = \{0\}$, then all possible selections are valid factorizations; propagate all of these factorizations to the recursion level above.

Otherwise apply μ to α. Search for a periodic block in $\mu(\alpha)$. Here, consider only blocks that do not contain any element multiple times. When such a periodic block is found, perform the recursive step as usual (with splitting the block first in the non-subgroup case).

As the recursive invocation now possibly supplies multiple factorizations instead of only one when returning, all of these factorizations are lifted and completed appropriately (just like with the single factorization as before, but with all factorizations now).

This algorithm is only successful when on the first recursion levels periodic blocks are found (until reaching $G = \{0\}$). In later sections we will prove for some groups and factorizations that this is the case and thus Algorithm 8.28 can be used for these.

The following theorem is from [Sza04] (Theorem 4.3.1). We will use it in later sections.

Theorem 8.29. *Let G be a cyclic group or a group of type $(p^u, p, \ldots, p)$, where p is a prime. Then in each multiple factorization of G into cyclic subsets at least one of the factors is a subgroup of G.*

If G is an abelian group which is not cyclic and not of type $(p^u, p, \ldots, p)$, then G has a multiple factorization by non-subgroup cyclic subsets.

8.7. Generic Factorization Algorithm

We now present a new generic factorization algorithm for block sequences of elements in abelian groups. In contrast to our previous algorithms, the generic factorization algorithm works with all block sequences, not just logarithmic signatures or multiple factorizations. For a few special cases, we prove that the algorithm is efficient.

Algorithm 8.30. Let H be an abelian group (represented as $H = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$) and $\beta \in \Xi_r(H)$ canonical (we do not require β to be a logarithmic signature or multiple factorization of H).

The following recursive algorithm computes all factorizations of a given element $h \in H$ with respect to β (if no such factorizations exist, an empty set is returned), when calling `FactorizeAbEx`(β, H, h).

Function `FactorizeAbEx`(BlockSequence $\alpha = (A_1, \ldots, A_n)$, Group G, GroupElement g) : Set<UInt[]>

1. If $n = 1$: `return` $\{(v) \mid 1 \leq v \leq |A_1|, A_1[v] = g\}$.
2. If there exists a block A_i in α with $|A_i| > |E(A_i)|$ (i.e. A_i contains an element at least twice):
   a) Let α' be the block sequence obtained from α by replacing block A_i by $E(A_i)$.
   b) Compute $L \leftarrow$ `FactorizeAbEx`(α', G, g).
   c) Let $W \leftarrow \{(v_1, v_2, \ldots, v_{i-1}, j, v_{i+1}, v_{i+2}, \ldots, v_l) \mid (v_1, v_2, \ldots, v_l) \in L, 1 \leq j \leq |A_i|\}$.
   d) `Return` $\{(v_1, v_2, \ldots, v_l) \in W \mid \alpha[1][v_1] + \alpha[2][v_2] + \ldots + \alpha[l][v_l] = g\}$.
3. If there exists a block A_i in α with $|A_i| = 1$:
   a) Let α' be the block sequence obtained from α by removing block A_i.
   b) Compute $L \leftarrow$ `FactorizeAbEx`(α', G, g).
   c) `Return` $\{(v_1, v_2, \ldots, v_{i-1}, 1, v_i, v_{i+1}, \ldots, v_l) \mid (v_1, v_2, \ldots, v_l) \in L\}$.
4. Compute $N \leftarrow$ `ChooseNormalSubgroup`(α, G). Let $\varphi : G \to G/N : x \mapsto x + N$.
5. Compute $L \leftarrow$ `FactorizeAbEx`$(\varphi(\alpha), G/N, \varphi(g))$.
6. `Return` $\{(v_1, v_2, \ldots, v_l) \in L \mid \alpha[1][v_1] + \alpha[2][v_2] + \ldots + \alpha[l][v_l] = g\}$.

Function `ChooseNormalSubgroup`(BlockSequence $\alpha = (A_1, \ldots, A_n)$, Group G) : $\mathcal{P}(G)$

1. Let $C \leftarrow$ `GetLowOrderElements`(α, G).
2. Let $\varphi_g : G \to G/\langle g \rangle : x \mapsto x + \langle g \rangle$.
   Compute $R \leftarrow \{(g, \mathrm{ord}(g) \cdot u, s) \mid g \in C, (u, s) =$ `EstSeqComplexity`$(\varphi_g(\alpha), G/\langle g \rangle)\}$.
   During the computation of the set R: if any g results in an estimated complexity of $(1, 1)$, immediately `return` $\langle g \rangle$.
3. Let $\eta \leftarrow \min\{u \mid (g, u, s) \in R\}$. Set $R \leftarrow \{(g, u, s) \in R \mid u = \eta\}$.
4. Let $\vartheta \leftarrow \min\{s \mid (g, u, s) \in R\}$. Set $R \leftarrow \{(g, u, s) \in R \mid s = \vartheta\}$.

5. Randomly pick one $(g, u, s) \in R$ and `return` $\langle g \rangle$.

Function `GetLowOrderElements`(BlockSequence $\alpha = (A_1, \ldots, A_n)$, Group G) : $\mathcal{P}(G)$

1. Let $\Psi(A_i) := \{A_i[k] - A_i[j] \mid 1 \leq j < k \leq |A_i|\}$ and $M := \bigcup_{i=1}^{n} \Psi(A_i)$.
2. Let $\check{q} \leftarrow \max\{|A_i| \mid 1 \leq i \leq n\}$. Set $d \leftarrow 0$ and $C \leftarrow \emptyset$.
3. While $C = \emptyset$:
   a) Set $d \leftarrow d + 1$.
   b) Set $C \leftarrow \left\{g \in M \mid \check{q}^{d-1} < \operatorname{ord}(g) \leq \check{q}^{d}\right\}$.
4. Interpret C as a list (order of elements arbitrary). For $x \leftarrow |C|$ down to 1:
   - If there exists a $1 \leq y < x$ with $\operatorname{ord}(C[y]) = \operatorname{ord}(C[x])$ and $C[y] \in \langle C[x] \rangle$: remove the xth element from C.
5. `Return` C.

Function `EstSeqComplexity`(BlockSequence $\alpha = (A_1, \ldots, A_n)$, Group G) : $\mathbb{N}^2$

1. Let $\alpha' = (A'_1, \ldots, A'_k) \leftarrow$ `SimplifySeq`(α, G).
2. If $k \leq 1$: `return` $(1, 1)$.
3. Let $u \leftarrow \prod_{i=1}^{k} |A'_i|$ and $s \leftarrow \sum_{i=1}^{k} \sum_{j=1}^{|A'_i|} \operatorname{ord}(A'_i[j])$.
4. For each $g \in$ `GetLowOrderElements`(α', G):
   a) Let $\varphi : G \to G/\langle g \rangle : x \mapsto x + \langle g \rangle$.
   b) Let $\gamma = (B_1, \ldots, B_l) \leftarrow$ `SimplifySeq`$(\varphi(\alpha'), G/\langle g \rangle)$.
   c) If $\operatorname{ord}(g) \cdot \prod_{i=1}^{l} |B_i| \leq u$: `return EstSeqComplexity`$(\gamma, G/\langle g \rangle)$.
5. `Return` (u, s).

Function `SimplifySeq`(BlockSequence $\alpha = (A_1, \ldots, A_n)$, Group G) : BlockSequence

1. Let α' be the block sequence obtained from α by replacing each block A_i by $E(A_i)$.
2. Remove all blocks A'_i with $|A'_i| = 1$ from α'.
3. `Return` α'.

Comments.

- We would like to note again that the hypothesis that α is canonical is not really a restriction; see Proposition 7.1.

- Recall that by Section 8.4 we can efficiently compute in all factor groups that can occur.

  The algorithm definition is rather mathematical (in order to make the core ideas clear). When implementing the algorithm, the approaches in Section 8.4 can be used. For example, for $\varphi(\alpha)$ (with $\varphi : G \to G/N : x \mapsto x + N$), we actually do not perform any computation with α, but instead append N to a normal subgroup chain, like in Algorithm 8.16.

  Element orders can for instance be computed using Lemma 11.6, or using the following: let $x = (x_1, x_2, \ldots, x_m) \in G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$, then

  $$\mathrm{ord}(x) = \mathrm{lcm}(\frac{p_1^{k_1}}{\gcd(x_1, p_1^{k_1})}, \frac{p_2^{k_2}}{\gcd(x_2, p_2^{k_2})}, \ldots, \frac{p_m^{k_m}}{\gcd(x_m, p_m^{k_m})})$$

  (which clearly can be computed efficiently).

- Observe that `FactorizeAbEx` and `EstSeqComplexity` are non-branching recursions; they call themselves at most once. The depth is polynomial in $\log |G|$.

- Let $\alpha = (A_1, A_2, \ldots, A_n)$. Define $\varkappa(\alpha, G, g)$ to be the number of different factorizations of g with respect to α, i.e. let

  $$\varkappa(\alpha, G, g) := |\{(i_1, i_2, \ldots, i_n) \in \mathbb{N}^n \mid 1 \le i_1 \le |A_1|, 1 \le i_2 \le |A_2|, \ldots, 1 \le i_n \le |A_n|, \\ A_1[i_1] + A_2[i_2] + \ldots + A_n[i_n] = g\}|.$$

  Let $\widehat{\varkappa}(\alpha, G, g)$ be the maximum number of factorizations returned by a `FactorizeAbEx` call on any recursion level when being called with the parameters α, G and g.

  Visually, when imagining the whole process as a recursive chain from top to bottom (it is non-branching, i.e. not a tree, just a chain) and arranging the returned factorizations horizontally (where each factorization is interpreted as an atom) on each recursion level, $\widehat{\varkappa}(\alpha, G, g)$ is the maximum width of the drawing.

  Obviously, $\widehat{\varkappa}(\alpha, G, g) \le \prod_{i=1}^{n} |A_i|$ (and later we give an example where "=" holds), but typically $\widehat{\varkappa}(\alpha, G, g)$ is much smaller.

  Examples.

  – Let $G = \mathbb{Z}_2 \oplus \mathbb{Z}_3 \oplus \mathbb{Z}_5$,

  $$\alpha = (A_1, A_2, A_3) = \begin{pmatrix}(0,0,0)\\(1,0,0)\end{pmatrix} \begin{pmatrix}(0,0,0)\\(0,1,0)\\(1,2,0)\end{pmatrix} \begin{pmatrix}(0,0,0)\\(1,1,1)\\(0,2,2)\\(1,2,3)\\(0,1,4)\end{pmatrix} \in \Lambda(G)$$

and $g \in G$. The algorithm will first move into the factor group G/A_1, i.e. factor $g + A_1$ with respect to (A_2, A_3) in G/A_1. Finally, it will look up $g + A_1 + A_2$ with respect to (A_3) in $G/(A_1 + A_2)$.

Exactly one element will be found for $g + A_1 + A_2$ with respect to (A_3) in $G/(A_1 + A_2)$, i.e. one index c (with $A_3[c] = g + A_1 + A_2$ in $G/(A_1 + A_2)$) is passed up to the caller; so

$$\varkappa((A_3), G/(A_1 + A_2), g + A_1 + A_2) = 1.$$

Before the second block was compressed (from $((0,0,0),(0,0,0),(0,0,0))$ to $((0,0,0)))$ and removed (because $((0,0,0))$ contains only one element), it was $((0,0,0),(0,0,0),(0,0,0))$. Thus, the factorization that comes up from the recursion level below is now expanded to three factorizations; so

$$\varkappa((((0,0,0)), A_3), G/(A_1 + A_2), g + A_1 + A_2) = 1,$$
$$\varkappa((((0,0,0),(0,0,0),(0,0,0)), A_3), G/(A_1 + A_2), g + A_1 + A_2) = 3.$$

There is exactly one index b such that $A_2[b] + A_3[c] = g + A_1$ in G/A_1, i.e. one factorization (b, c) is passed up to the caller; so

$$\varkappa((A_2, A_3), G/A_1, g + A_1) = 1.$$

Before the first block was compressed (from $((0,0,0),(0,0,0))$ to $((0,0,0)))$ and removed (because $((0,0,0))$ contains only one element), it was $((0,0,0),(0,0,0))$. Thus, the factorization that comes up from the recursion level below is now expanded to two factorizations; so

$$\varkappa((((0,0,0)), A_2, A_3), G/A_1, g + A_1) = 1,$$
$$\varkappa((((0,0,0),(0,0,0)), A_2, A_3), G/A_1, g + A_1) = 2.$$

There is exactly one index a such that $A_1[a] + A_2[b] + A_3[c] = g$ in G, i.e. the algorithm finally returns one factorization (a, b, c); so

$$\varkappa(\alpha, G, g) = 1.$$

The maximum of the $\varkappa$ values is 3, i.e. $\widehat{\varkappa}(\alpha, G, g) = 3$.

– Let $G = \mathbb{Z}_3$ and $\alpha = ((0,1,1))$ (one block; α is a block sequence, no logarithmic signature). Then $\varkappa(\alpha, G, g) = \widehat{\varkappa}(\alpha, G, g)$ for all $g \in G$, $\widehat{\varkappa}(\alpha, G, 0) = 1$, $\widehat{\varkappa}(\alpha, G, 1) = 2$ and $\widehat{\varkappa}(\alpha, G, 2) = 0$.

The run-time of the algorithm mainly depends on $\widehat{\varkappa}(\alpha, G, g)$. If $\widehat{\varkappa}(\alpha, G, g)$ is polynomial in $\ell(\alpha)$, then the algorithm is efficient. If $\widehat{\varkappa}(\alpha, G, g)$ is exponential, exponentially many elements need to be passed to the caller on some recursion level, which requires exponential time.

- We analyze the effects of the operations on the run-time.

– The removal of duplicate elements in blocks simplifies the block sequence without influencing $\widehat{\varkappa}(\alpha, G, g)$. In other words, the maximum value $\widehat{\varkappa}(\alpha, G, g)$ is always caused by a different operation, not the removal of duplicate elements.

  Let $\alpha = (A_1, A_2, \ldots, A_n)$ be a block sequence in which A_1 contains duplicate elements. Define $\alpha' := (E(A_1), A_2, \ldots, A_n)$. Then clearly $\varkappa(\alpha, G, g) \geq \varkappa(\alpha', G, g)$, i.e. equally many or more elements are passed up from the upper recursion level for α than from the lower level for α'.

  The relationship can be stated more precisely. For all $a \in E(A_1)$, let $m_a(\alpha', G, g)$ be the number of factorizations of g with respect to α' where a is selected in the first block. Then

$$\varkappa(\alpha, G, g) = \sum_{a \in E(A_1)} \mathfrak{c}_a(A_1) \cdot m_a(\alpha', G, g).$$

– The removal of blocks of size 1 does not influence $\widehat{\varkappa}(\alpha, G, g)$.

  Let $\alpha = (A_1, A_2, \ldots, A_n)$ with $|A_1| = 1$ (note that by design this implies $A_1 = \{0\}$), then clearly $\varkappa(\alpha, G, g) = \varkappa((A_2, A_3, \ldots, A_n), G, g)$.

– At the end of the `FactorizeAbEx` method, a normal subgroup N is chosen and we move into the factor group G/N. Let $\varphi : G \to G/N : x \mapsto x + N$.

  * If $\alpha \in \Lambda(G)$, then

$$\varkappa(\varphi(\alpha), G/N, \varphi(g)) = |N| \cdot \varkappa(\alpha, G, g).$$

  * If α is an arbitrary block sequence, the effect may vary, but clearly

$$\varkappa(\varphi(\alpha), G/N, \varphi(g)) \geq \varkappa(\alpha, G, g).$$

  So, this operation may influence $\widehat{\varkappa}(\alpha, G, g)$.

Observe that the $\varkappa$ values may increase and decrease while descending recursively (moving into a factor group increases it, removing duplicate elements decreases it). Thus, $\widehat{\varkappa}(\alpha, G, g)$ is determined on any of the recursion levels, not necessarily on the top or bottom one. However, it is always determined by a movement into a factor group.

The multiplicities of factorizations caused by moving into factor groups do not necessarily accumulate (multiplicatively), because between the movements into factor groups duplicate elements are removed. As an example, we would like to highlight one important special situation. Let $\alpha = (A_1, A_2, \ldots, A_n)$ be a multiple factorization (with multiplicity ≥ 1) of G with $A_1 \leq G$, $N = A_1$ and $\varphi : G \to G/N : x \mapsto x + N$. We have

$$\varkappa(\varphi(\alpha), G/N, \varphi(g)) = |N| \cdot \varkappa(\alpha, G, g).$$

In $\varphi(\alpha)$ the first block contains only $(0,\ldots,0)$ elements, thus it is compressed to a single $(0,\ldots,0)$ element. Then, the block containing only $(0,\ldots,0)$ is removed. Define $\alpha' := (\varphi(A_2), \varphi(A_3), \ldots, \varphi(A_n))$ (clearly α' is a multiple factorization of G/N).

$$\begin{aligned}
\varkappa(\alpha, G, g) &= \frac{|N| \cdot \varkappa(\alpha, G, g)}{|N|} = \frac{\varkappa(\varphi(\alpha), G/N, \varphi(g))}{|N|} \\
&= \frac{\varkappa(\varphi(\overbrace{(((0,\ldots,0),(0,\ldots,0),\ldots,(0,\ldots,0))}^{|A_1|}, A_2, \ldots, A_n)), G/N, \varphi(g))}{|N|} \\
&= \varkappa(\varphi((((0,\ldots,0)), A_2, \ldots, A_n)), G/N, \varphi(g)) \\
&= \varkappa(\alpha', G/N, \varphi(g)).
\end{aligned}$$

As we see, during the process locally the number of factorizations increases by a factor of $|N|$, but overall this does not accumulate. When g has t factorizations with respect to α in G, $\varphi(g)$ also has t factorizations with respect to α' in G/N. Thus, the movement into the factor group G/N basically has no effect on the run-time of the algorithm. $\widehat{\varkappa}(\alpha, G, g)$ is polynomial in $\ell(\alpha)$ if and only if $\varkappa(\alpha', G/N, \varphi(g))$ is polynomial in $\ell(\alpha)$.

Obviously, when the algorithm moves into a factor group where no block of the same size vanishes completely, the number of factorizations does not necessarily decrease immediately again and the movement into the factor group *can* contribute to an exponential run-time.

In general, let $g \in G$ be a candidate element returned by `GetLowOrderElements` with a reasonably low order for a block sequence $\alpha = (A_1, A_2, \ldots, A_n)$ in G, and $\varphi : G \to G/\langle g\rangle : x \mapsto x + \langle g\rangle$. Moving into the factor group $G/\langle g\rangle$ basically has no effect on the run-time of the algorithm if

$$\operatorname{ord}(g) \cdot \prod_{i=1}^{n} |E(\varphi(A_i))| \leq \prod_{i=1}^{n} |E(A_i)|.$$

In contrast, if "$>$" holds, the movement into the factor group $G/\langle g\rangle$ can contribute to an exponential run-time of the algorithm.

- The `EstSeqComplexity` function tries to estimate the complexity of a block sequence. For shortness, write $\xi(\alpha, G) :=$ `EstSeqComplexity`(α, G).

  The function realizes the following:

  - The estimated complexity of a block sequence where a block contains duplicate elements should be the same as when the duplicate elements are merged to one.

    $$\xi((A_1, A_2, \ldots, A_n), G) = \xi((E(A_1), E(A_2), \ldots, E(A_n)), G).$$

  - If the block sequence $\alpha = (A_1, A_2, \ldots, A_n)$ contains a block of size 1 (w.l.o.g. let $|A_1| = 1$), then the estimated complexity of α should be the same as the

one of the block sequence where A_1 is removed.

$$\xi((A_1, A_2, \ldots, A_n), G) = \xi((A_2, A_3, \ldots, A_n), G) \quad (\text{when } |A_1| = 1).$$

– As stated above, moving into a factor group $G/\langle g\rangle$ (let $\varphi : G \to G/\langle g\rangle : x \mapsto x + \langle g\rangle$) has no effect on the run-time if g has a reasonably low order and

$$\operatorname{ord}(g) \cdot \prod_{i=1}^{n} |E(\varphi(A_i))| \leq \prod_{i=1}^{n} |E(A_i)|.$$

Thus if such a g exists (i.e. returned by `GetLowOrderElements` with low order and fulfilling the above inequality), the estimated complexity of α should be the same as the one of the reduced block sequence for the factor group.

$$\xi(\alpha, G) = \xi(\varphi(\alpha), G/\langle g\rangle).$$

– If none of the above applies, we estimate the complexity using two natural numbers.

∗ It primarily rates the block sequence by the number of elements that are generated (counting duplicates).

This is a worst-case assumption. Let α be a block sequence generating c elements. Then $\max\{\varkappa(\alpha, G, g) \mid g \in G\} \leq c$ (these at most c factorizations can be enumerated using brute-force).

∗ The secondary rating value is the sum of the orders of all elements in the block sequence.

Low order elements indicate a simpler structure, and more elements are candidates for generating candidate subgroups in the future.

The `EstSeqComplexity` function structurally is similar to the `FactorizeAbEx` function. Both recursively simplify the input block sequence and move into factor groups. When no other simplifications work, `FactorizeAbEx` moves into non-optimal factor groups (i.e. where the reduced block sequence for the factor group is comparatively large). In contrast, `EstSeqComplexity` does not do this: when reasonably low order candidates are available, it only moves into factor groups that cannot contribute to an exponential run-time of the algorithm.

• The function `ChooseNormalSubgroup` has the task to choose a "good" normal subgroup N.

As candidates for N, the cyclic subgroups generated by single elements returned by `GetLowOrderElements` are evaluated. These elements have a relatively low order and guarantee a simplification; details can be found below.

Goals of `ChooseNormalSubgroup` are the following:

– $|N|$ should be rather small.

If $|N|$ is large, many factorizations may need to be passed up from the recursion level below, which results in a large run-time. Thus, the smaller $|N|$ the better.

– Choosing an N can result in various simplifications being possible in the factor group. For example, when choosing an N, a block might not vanish immediately, but it might become periodic and after moving into the factor group induced by the subgroup block of periods there might be another periodic block, and so on. The `EstSeqComplexity` function takes account of this.

In order to obtain the final complexity rating $\mathfrak{r}$, the primary rating returned by the `EstSeqComplexity` function is multiplied by $|N|$. This accounts for the investment of moving into the factor group G/N (the `EstSeqComplexity` function just rates a given block sequence; it does not know how much we would need to invest to obtain the block sequence and factor group). For example, when a candidate element of order 8 only results in a block of size 4 vanishing (the primary rating in G/N would be $\frac{1}{4}$th of the rating in G), this is just as good as a candidate element of order 4 only resulting in a block of size 2 vanishing (the primary rating in G/N would be $\frac{1}{2}$th of the rating in G); $8 \cdot \frac{1}{4} = 4 \cdot \frac{1}{2}$.

The lower the rating $\mathfrak{r}$ for the block sequence in G/N the better.

Observe that these goals are contradictory. For example, when $N = G$, the rating will be minimal, but the run-time is large, because $|N|$ is large. Thus, we need to prioritize the goals.

The algorithm prioritizes keeping $|N|$ reasonably small. In the first iteration of the "While" loop of `GetLowOrderElements`, elements with a maximum order of $\check{q}$ (which is the size of the largest block in α) are regarded as candidates for spanning N. Only if no candidates with such orders are found at all, the algorithm continues by searching for candidates again, now accepting orders between $\check{q}$ and $\check{q}^2$. This continues until at least one candidate has been found (the input α is guaranteed to contain at least one non-$(0, \ldots, 0)$ element).

When having found candidates with reasonable orders, the candidates are filtered by the ratings they induce in their factor groups. The higher each rating is, the more complex the block sequence is considered. The `ChooseNormalSubgroup` function first filters candidates by the first rating value and then by the second (i.e. the first rating is considered to be more important; only when the first ratings of two candidates are the same, the second rating decides which candidate is chosen).

Finally, there might still be multiple candidates left. The algorithm randomly chooses one of them.

- The `GetLowOrderElements` function returns elements that have a relatively low order and guarantee a simplification when moving into factor groups induced by the cyclic subgroups generated by the elements. The order filtering has been discussed already; we now show how the initial candidates are chosen.

A candidate element g must result in an immediate simplification when moving into the factor group $G/\langle g\rangle$. Such an immediate simplification occurs when there are duplicate elements in a block. Thus for a candidate element g (let $\varphi_g : G \to G/\langle g\rangle : x \mapsto x + \langle g\rangle$),

$$\prod_{i=1}^{n} |E(\varphi_g(A_i))| < \prod_{i=1}^{n} |E(A_i)|$$

must hold.

Elements fulfilling this inequality can be found easily. For example, all elements in $E(\alpha) \setminus \{(0,\ldots,0)\}$ fulfill it, because for all $g \in E(\alpha) \setminus \{(0,\ldots,0)\}$ the element g in one of the blocks becomes a duplicate of $(0,\ldots,0)$ when moving into the factor group $G/\langle g\rangle$. However, there are more. We do not necessarily need to create duplicates of $(0,\ldots,0)$; arbitrary elements may become duplicates. Clearly, for every $N \leq G$ and $a, b \in G$ we have $a + N = b + N \Leftrightarrow b - a \in N$. With $N := \langle b - a\rangle$ for some $a, b \in A_i$, the elements a and b become duplicates in G/N. As $\langle b - a\rangle = \langle a - b\rangle$, `GetLowOrderElements` enumerates the candidates in one order only (see definition of $\Psi(A_i)$; only the element with a lower index is inverted).

Finally, some elements may generate exactly the same subgroup. These are filtered out at the end of the `GetLowOrderElements` function.

Comparison. The new Algorithm 8.30 supersedes all previously mentioned factorization algorithms for abelian groups. This might not be clear immediately (the new algorithm is rather different, it does not search for periodic blocks, etc.). Thus we now show that the new algorithm works efficiently whenever the previous algorithms worked efficiently.

- Most previous algorithms required the existence of a subgroup block on each recursion level. Let $\alpha = (A_1, A_2, \ldots, A_n)$ be such a block sequence, i.e. let α be a block sequence such that w.l.o.g. $A_1 \leq G$, $A_2 \leq G/A_1$, $A_3 \leq G/(A_1 + A_2)$, and so on. Furthermore, let $|A_i| = |E(A_i)|$ for all $1 \leq i \leq n$.

  As $A_1 \leq G$, we have $\mathrm{ord}(a) \leq |A_1|$ for all $a \in A_1$, i.e. elements that are generating A_1 are returned by `GetLowOrderElements`. Let $a \in A_1 \setminus \{(0,\ldots,0)\}$.

  - If $\langle a\rangle = A_1$, the whole block A_1 vanishes, keeping the run-time of the algorithm low.
  - Let $\langle a\rangle \neq A_1$. Clearly $\langle a\rangle \leq A_1$. Define $\varphi_a : G \to G/\langle a\rangle : x \mapsto x + \langle a\rangle$. Then $\varphi_a(A_1) \leq G/\langle a\rangle$ (homomorphic images of subgroups are subgroups) and $|E(\varphi_a(A_1))| = \frac{|A_1|}{\mathrm{ord}(a)}$.

    So, when we move into the factor group $G/\langle a\rangle$ (which is optimal, because the size of A_1 reduces to $\frac{1}{\mathrm{ord}(a)}$th of its size), $\varphi_a(A_1)$ again is a subgroup in $G/\langle a\rangle$. Iterating this idea we observe that the original block A_1 is deconstructed completely (step by step) before `EstSeqComplexity` returns.

It remains to show that the order in which the candidate elements are picked is irrelevant. Let $g \in G$ be the element being picked (not necessarily $g \in A_1$). The picking order irrelevance immediately follows from the fact that $\varphi_g(A_1) \leq G/\langle g \rangle$ holds for all g, i.e. A_1 is always deconstructed completely.

In total we get

$$\widehat{\varkappa}(\alpha, G, g) \leq \max\{|A_i| \mid 1 \leq i \leq n\},$$

i.e. the new algorithm works very efficiently.

- Similarly the new algorithm works efficiently when a periodic block is found on each recursion level. The subgroups of periods within periodic blocks are deconstructed completely.

  Just like in the subgroups situation, in total we again get

  $$\widehat{\varkappa}(\alpha, G, g) \leq \max\{|A_i| \mid 1 \leq i \leq n\},$$

  i.e. the new algorithm works very efficiently.

In contrast to the previous algorithms, which only worked for logarithmic signatures or multiple factorizations with special properties, the new Algorithm 8.30 works for *all* block sequences ("working" in the sense that the algorithm does output all factorizations of a given element, just its run-time may vary).

Run-time examples.

- Let $\alpha = (A_1, A_2, \ldots, A_n) \in \Lambda(G)$ canonical with $|A_i| \in \mathbb{P}$ for all $1 \leq i \leq n$. Then the algorithm works efficiently.

  By Rédei's theorem (Theorem 8.1) there is a subgroup block on each recursion level, thus as previously shown $\widehat{\varkappa}(\alpha, G, g) \leq \max\{|A_i| \mid 1 \leq i \leq n\}$.

- Let G be a p-group of order $|G| = p^n$ (with $p \in \mathbb{P}_{\geq 3}$) and $\alpha = (A_1, A_2, \ldots, A_k) \in \Lambda(G)$ an aperiodic logarithmic signature generated using Algorithm 6.4. The factorization algorithm works efficiently for α.

  It is easy to see why this is true. Let $B := A_1 + A_2$. By construction of Algorithm 6.4, $B \leq G$ and $B \cong \mathbb{Z}_p^4$, thus $\{\mathrm{ord}(g) \mid g \in B\} \subseteq \{1, p\}$. So, on the topmost recursion levels of the factorization algorithm, `GetLowOrderElements` returns the elements of A_1 and A_2. Consequently, after the first recursion levels we are in a factor group G/N with $B \leq N$. Now there always exists a periodic block in each factor group on all further recursion levels.

  More generally, let $\beta \in \Lambda(H)$ be an aperiodic logarithmic signature for a p-group H and α an aperiodic logarithmic signature for a group $G \geq H$ generated by extending β using any of the lemmas in Section 6.4.1. Then the factorization algorithm is efficient, because factoring with respect to α is as most as hard as with respect to β.

- Experiments show that the factorization algorithm probably works efficiently for all abelian logarithmic signatures created by LS-Gen (Section 6.5). Intuitively, LS-Gen has sufficiently powerful transformations to create logarithmic signatures that could maybe resist the factorization algorithm. However, parameters used in practice do not seem to provide sufficient confusion for the factorization algorithm. For example, a block substitution can transform a logarithmic signature having a periodic block into an aperiodic logarithmic signature, but practically the number of blocks involved in this operation is limited to a very low value (otherwise the block substitution transformation cannot be computed efficiently) and the number of rounds is rather limited; and as we have seen in the previous example such small, local aperiodic obstacles are not sufficient to result in an exponential factorization run-time.

- Let $G = \mathbb{Z}_{2^n}$ and $\alpha = \binom{0}{1}\binom{0}{1}\cdots\binom{0}{1} \in \Xi_n(G)$ (note that α is just a pseudo-logarithmic signature, not a logarithmic signature). Assume that we want to find the factorization of the element n (there is exactly one factorization for this element, so theoretically an algorithm can return the answer in polynomial time).

  For our factorization algorithm the only candidate element for a subgroup is 1, i.e. $N = G$. From the second recursion level, 2^n solutions need to be passed up. So, the run-time of the algorithm is exponential in $\ell(\alpha)$.

Implementation. We have implemented this algorithm. See Section 12.3 for the command line syntax.

Optimizations. The description of the algorithm prefers clarity over performance. There are various optimizations possible, including but not limited to:

- Algorithm 8.13 (for element equality tests in factor groups) works for all factor groups that are described by a chain of normal subgroups. In Algorithm 8.30, each of these normal subgroups is cyclic, which can be used to improve the performance.

- For many group elements, the order is computed multiple times (for example, in `GetLowOrderElements` the orders of all returned elements are computed and they are computed again in `EstSeqComplexity`). Caching the order (storing the order along with the group element) improves the performance.

- Observe that during `ChooseNormalSubgroup` and `EstSeqComplexity`, a loop iterates over all elements returned by `GetLowOrderElements`, and for every element it may happen that the function returns immediately, i.e. in this case the remaining elements returned by `GetLowOrderElements` are unnecessary. Thus, implementing `GetLowOrderElements` as an iterator (i.e. in such a way that it computes a returned element only when the calling code requests it; e.g. in C# using `yield return`) can improve the performance. The last part of `GetLowOrderElements` (where the set C is filtered) is very suited for this.

Non-abelian groups. When trying to extend this algorithm for non-abelian groups, the main obstacle is to find a way how to efficiently compute in factor groups. Furthermore,

it is unclear how factor groups should be chosen (a cyclic subgroup generated by an element is not necessarily normal in a non-abelian group).

Specific run-time analysis. As described above, the run-time of the algorithm highly depends on the structure of the input. Anyway, we want to show the run-time in one specific situation:

Proposition 8.31. *Let H be an abelian group (represented as $H = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$) and $\beta = (B_1, B_2, \ldots, B_r) \in \Lambda(H)$ canonical with $|B_i| \in \mathbb{P}$ for all $1 \leq i \leq r$. Let $q := \max\{|B_i| \mid 1 \leq i \leq n\}$.*

Then Algorithm 8.30 computes the factorization of any group element using $O(r^6 \cdot q^6 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot \ell(\beta))$ group operations.

Proof. We assume that Algorithm 8.30 uses the same approach as Algorithm 8.16 to compute in factor groups. A group element addition (in any factor group) requires one operation in H (simply adding representatives), and a group element equality test (in any factor group) requires $O(m \cdot r \cdot \ell(\beta))$ operations in H (note that this holds in this specific case only; in general, without further optimizations, the normal subgroup chain may be larger and thus element equality tests in factor groups may require more group operations in H).

For computing the order of a group element, we assume that the approach of Lemma 11.6 is used, i.e. we need $O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta))$ group operations for computing the order of any group element in any factor group.

We now determine the number of group operations performed by each function.

- In the function `SimplifySeq`, for the replacement of each block A_i by $E(A_i)$ there are at most $\sum_{i=1}^{n} \frac{(|A_i|-1)\cdot|A_i|}{2} < n \cdot q^2$ factor group operations required, i.e. $O(n \cdot q^2 \cdot m \cdot r \cdot \ell(\beta))$ operations in H. We assume that removing all blocks A'_i with $|A'_i| = 1$ from α' requires no group operations.

- In `GetLowOrderElements`, the set $\Psi(A_i)$ contains at most $\widehat{q} := \frac{(q-1)\cdot q}{2}$ elements, thus $|M| \leq n \cdot \widehat{q}$. In order to filter any duplicate elements from M, we need at most $\frac{(n\cdot\widehat{q}-1)\cdot n\cdot\widehat{q}}{2}$ group element equality tests (each requiring $O(m \cdot r \cdot \ell(\beta))$ operations in H).

  When constructing C, for each element in M an element order computation is performed. Due to $|C| \leq |M|$, we obtain $|C| \leq n \cdot \widehat{q}$.

  In the next loop, x runs over $|C|$ different values, and in each loop y runs up to x (exclusively). $\mathrm{ord}(C[x])$ and $\langle C[x] \rangle$ can be computed in the outer loop (where x is the running variable). $\langle C[x] \rangle$ is computed explicitly (note that due to the structure of β we have $|\langle C[x] \rangle| \leq q$; and as we have previously computed $\mathrm{ord}(C[x])$, computing $\langle C[x] \rangle$ requires at most $q - 1$ group element additions in H).

In total, `GetLowOrderElements` requires

$$\underbrace{n \cdot \widehat{q}}_{\text{Build } M} + \underbrace{\frac{(n \cdot \widehat{q} - 1) \cdot n \cdot \widehat{q}}{2} \cdot O(m \cdot r \cdot \ell(\beta))}_{\text{Filter duplicates from } M} + \underbrace{n \cdot \widehat{q} \cdot O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta))}_{\text{Build } C} +$$

$$\sum_{x=1}^{|C|} \Big(\underbrace{O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta))}_{\text{Compute ord}(C[x])} + \underbrace{q-1}_{\text{Compute } \langle C[x] \rangle} +$$

$$\sum_{y=1}^{x-1} \big(\underbrace{O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta))}_{\text{Compute ord}(C[y])} + \underbrace{q \cdot O(m \cdot r \cdot \ell(\beta))}_{\text{Test } C[y] \in \langle C[x] \rangle} \big) \Big)$$

$$\begin{aligned} &= O(n^2 \cdot q^4 \cdot m \cdot r \cdot \ell(\beta)) + O(n \cdot q^2 \cdot (\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta)) + \\ &\quad O(n^2 \cdot q^4) \cdot \big(O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta)) + O(q \cdot m \cdot r \cdot \ell(\beta)) \big) \\ &= O(n^2 \cdot q^4 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta)) \end{aligned}$$

group operations (in H).

- Let $f(n)$ be the number of group operations (in H) that `EstSeqComplexity` requires for n blocks without the recursive call. We have

$$\begin{aligned} f(n) = \; & \underbrace{O(n \cdot q^2 \cdot m \cdot r \cdot \ell(\beta))}_{\text{First } \texttt{SimplifySeq}} + \underbrace{\ell(\alpha) \cdot O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta))}_{\text{Compute } s} + \\ & \underbrace{O(n^2 \cdot q^4 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta))}_{\texttt{GetLowOrderElements}} + \underbrace{O(n \cdot q^2)}_{\text{Low order el.}} \cdot \\ & \left(\underbrace{O(n \cdot q^2 \cdot m \cdot r \cdot \ell(\beta))}_{\text{Inner } \texttt{SimplifySeq}} + \underbrace{O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta))}_{\text{Compute ord}(g)} \right). \end{aligned}$$

For the total number of group operations (in H) that `EstSeqComplexity` requires at most, we obtain:

$$\begin{aligned} \sum_{i=1}^{n} f(i) = \; & \sum_{i=1}^{n} \Big(O(i \cdot q^2 \cdot m \cdot r \cdot \ell(\beta)) + \ell(\alpha_{(i)}) \cdot O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta)) + \\ & O(i^2 \cdot q^4 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta)) + O(i \cdot q^2) \cdot \\ & \big(O(i \cdot q^2 \cdot m \cdot r \cdot \ell(\beta)) + O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta)) \big) \Big) \\ \subseteq \; & O(n^2 \cdot q^2 \cdot m \cdot r \cdot \ell(\beta)) + n \cdot \ell(\alpha) \cdot O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta)) + \\ & O(n^3 \cdot q^4 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta)) + O(n^2 \cdot q^2) \cdot \\ & \big(O(n \cdot q^2 \cdot m \cdot r \cdot \ell(\beta)) + O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta)) \big) \\ \overset{*}{=} \; & O(n^3 \cdot q^4 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta)). \end{aligned}$$

(*) $\ell(\alpha) \leq n \cdot q$.

- `ChooseNormalSubgroup` requires at most

$$\underbrace{O(n^2 \cdot q^4 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta))}_{\texttt{GetLowOrderElements}} + \underbrace{O(n \cdot q^2)}_{\text{Low order el.}} \cdot$$
$$\left(\underbrace{O((\log_2 |H|)^2 \cdot m \cdot r \cdot \ell(\beta))}_{\text{Compute ord}(g)} + \underbrace{O(n^3 \cdot q^4 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta))}_{\texttt{EstSeqComplexity}} \right) +$$
$$\underbrace{q-1}_{\langle g \rangle}$$
$$= O(n^4 \cdot q^6 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta))$$

  group operations (in H).

- Let $f(n)$ be the number of group operations (in H) that `FactorizeAbEx` requires for n blocks without any recursive call. For $n \geq 2$, we have

$$f(n) = \underbrace{O(n \cdot q^2 \cdot m \cdot r \cdot \ell(\beta))}_{\text{Compute } E(A_i) \text{ for all } i} + \underbrace{O(n^4 \cdot q^6 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta))}_{\texttt{ChooseNormalSubgroup}} +$$
$$\underbrace{O(q \cdot m \cdot n \cdot \ell(\beta))}_{\text{Completing fac.}}$$
$$= O(n^4 \cdot q^6 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta)).$$

  In total, `FactorizeAbEx` requires at most

$$\underbrace{q \cdot O(m \cdot r \cdot \ell(\beta))}_{\text{Case 1 block}} + \sum_{n=2}^{r} f(n)$$
$$= O(q \cdot m \cdot r \cdot \ell(\beta)) + \sum_{n=2}^{r} O(n^4 \cdot q^6 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta))$$
$$= O(r^5 \cdot q^6 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot r \cdot \ell(\beta))$$
$$= O(r^6 \cdot q^6 \cdot ((\log_2 |H|)^2 + q) \cdot m \cdot \ell(\beta))$$

  group operations (in H). □

8.8. Factoring by Combining Solutions in Factor Groups

8.8.1. Statically Chosen Factor Groups

Let $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$, $\alpha = (A_1, A_2, \ldots, A_n) \in \Lambda(G)$ canonical and $g \in G$. Define

$$\varphi_c : G \to \mathbb{Z}_{p_c^{k_c}} : (g_1, g_2, \ldots, g_m) \mapsto g_c$$

the projection on the cth component. φ_c is a homomorphism and $\varphi_c(\alpha)$ is a $\frac{|G|}{p_c^{k_c}}$-factorization of $\mathbb{Z}_{p_c^{k_c}}$. For shortness, write $\alpha_c := \varphi_c(\alpha)$. Observe that

$$\begin{aligned}
&\alpha[1][i_1] + \alpha[2][i_2] + \ldots + \alpha[n][i_n] = g \\
\Leftrightarrow\ &\varphi_c(\alpha[1][i_1] + \alpha[2][i_2] + \ldots + \alpha[n][i_n]) = \varphi_c(g) \text{ for all } 1 \leq c \leq m \\
\Leftrightarrow\ &\alpha_c[1][i_1] + \alpha_c[2][i_2] + \ldots + \alpha_c[n][i_n] = \varphi_c(g) \text{ for all } 1 \leq c \leq m.
\end{aligned}$$

Define $L(\alpha, g, c) := \{(i_1, i_2, \ldots, i_n) \in \mathbb{N}^n \mid \alpha_c[1][i_1] + \alpha_c[2][i_2] + \ldots + \alpha_c[n][i_n] = \varphi_c(g)\}$. Then $\bigcap_{c=1}^{m} L(\alpha, g, c)$ contains exactly one element, namely the factorization of g.

We now present an approach to turn this idea into a factorization algorithm. The main obstacle is that the sets $L(\alpha, g, c)$ are very large: $|L(\alpha, g, c)| = \frac{|G|}{p_c^{k_c}}$, too large to compute. One optimization is to introduce "do not care" indices for blocks that contain only identity elements in $\mathbb{Z}_{p_c^{k_c}}$. Define

$$\varpi(x, y) := \begin{cases} -1, & \text{if } |E(\alpha_c[x])| = 1, \\ y, & \text{otherwise,} \end{cases}$$

$$L'(\alpha, g, c) := \{(\varpi(1, i_1), \varpi(2, i_2), \ldots, \varpi(n, i_n)) \mid (i_1, i_2, \ldots, i_n) \in L(\alpha, g, c)\},$$

$$\theta : \mathbb{Z}^2 \to \{0, 1\} : (a, b) \mapsto \begin{cases} 1, & \text{if } a = -1 \text{ or } b = -1 \text{ or } a = b, \\ 0, & \text{otherwise,} \end{cases}$$

$$\widehat{\theta} : \mathbb{Z}^n \oplus \mathbb{Z}^n \to \{0, 1\} : ((i_1, i_2, \ldots, i_n), (j_1, j_2, \ldots, j_n)) \mapsto \prod_{x=1}^{n} \theta(i_x, j_x).$$

ϖ replaces the element index y by -1 if and only if the block with index x contains only identity elements in α_c. $L'(\alpha, g, c)$ is the set of factorizations where all the indices for blocks containing only identity elements are replaced by -1 (if such a replacement happens at least once, some factorization vectors become identical, i.e. $|L'(\alpha, g, c)| < |L(\alpha, g, c)|$). $\widehat{\theta}$ tests whether two factorizations are compatible, i.e. whether they can specify the same factorization of a group element. For this, every index must either be the same or at least one of the two indices must specify the "do not care" value -1; θ tests this for one index.

There exists exactly one factorization vector $u = (i_1, i_2, \ldots, i_n) \in \mathbb{N}^n$ for which there exist $v_1 \in L'(\alpha, g, 1), v_2 \in L'(\alpha, g, 2), \ldots, v_m \in L'(\alpha, g, m)$ with $\widehat{\theta}(u, v_k) = 1$ for all $1 \leq k \leq m$. u is the factorization of g.

Algorithm 8.32. Let $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$, $\alpha = (A_1, A_2, \ldots, A_n) \in \Lambda(G)$ canonical and $g \in G$ an element to be factored.

The sets $L'(\alpha, g, c)$ can be computed without computing the full sets $L(\alpha, g, c)$. Let α'_c be the block sequence obtained from α_c by removing all blocks that contain only identity elements. In the remaining block sequence, compute all factorizations of $\varphi_c(g)$, insert the value -1 at the positions of the removed blocks and put the resulting factorizations into $L'(\alpha, g, c)$.

Finally, find a factorization vector $u = (i_1, i_2, \ldots, i_n) \in \mathbb{N}^n$ that is compatible with all $L'(\alpha, g, c)$ sets (i.e. in every $L'(\alpha, g, c)$ there exists a vector that is compatible with u).

Various strategies are possible for finding u. We do not specify one in detail, because in the following we will see that independent of which strategy is used, the algorithm can have a high run-time.

Furthermore, note that it is not specified how to compute the factorizations of $\varphi_c(g)$ (but even when there is a very efficient way for this, the algorithm can have a high run-time anyway).

Run-time. Unfortunately this algorithm can have a high run-time even for very simple logarithmic signatures, because all $L'(\alpha, g, c)$ sets may be large.

We construct some example logarithmic signatures for which the algorithm requires exponential time. For the first construction, we prove a small lemma.

Lemma 8.33. *There exists an invertible matrix $M \in \mathcal{M}_n(\mathbb{Z}_2)$ such that M contains $n^2 - n + 1$ ones. Especially, every row of M contains at least $n - 1$ ones.*

Proof. We construct a matrix M. Let us start with the identity matrix I_n. The column vectors clearly are linear independent. Elementary row operations do not break the linear independency. We now add the first $n - 1$ rows onto the last row. Subsequently, we add the last row back onto the first $n - 1$ rows. The resulting matrix fulfills the requested property.

For example, for $n = 7$ we get:

$$I_7 \rightsquigarrow \begin{pmatrix} 1&0&0&0&0&0&0\\ 0&1&0&0&0&0&0\\ 0&0&1&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0\\ 1&1&1&1&1&1&1 \end{pmatrix} \rightsquigarrow \begin{pmatrix} 0&1&1&1&1&1&1\\ 1&0&1&1&1&1&1\\ 1&1&0&1&1&1&1\\ 1&1&1&0&1&1&1\\ 1&1&1&1&0&1&1\\ 1&1&1&1&1&0&1\\ 1&1&1&1&1&1&1 \end{pmatrix} =: M.$$

The resulting matrix M contains $n^2 - n + 1$ ones. Each of the first $n - 1$ rows contains $n - 1$ ones and the last row contains n ones.

This is the maximum number of ones that M can contain. A matrix containing only $n - 2$ zeros would contain the row $(1, 1, \ldots, 1)$ twice and thus would not be invertible anymore. □

Now back to the high run-time examples.

Example 8.34. Let $G = \mathbb{Z}_2^n$ and M a matrix as in Lemma 8.33. Let $M[i]$ denote the ith row vector in M. Clearly,

$$\alpha := \begin{pmatrix} (0,\ldots,0) \\ M[1] \end{pmatrix} \begin{pmatrix} (0,\ldots,0) \\ M[2] \end{pmatrix} \cdots \begin{pmatrix} (0,\ldots,0) \\ M[n] \end{pmatrix}$$

is a logarithmic signature for G.

The factorization Algorithm 8.32 has an exponential run-time for α, because for every c the block sequence α_c contains at least $n - 1$ blocks with two distinct elements and thus $|L'(\alpha, g, c)| \geq 2^{(n-1)-1} = 2^{n-2}$, which is exponential in $\ell(\alpha) = 2n$.

Example 8.35. Let $n \geq 2$, $G = \mathbb{Z}_{2^{2n}} \oplus \mathbb{Z}_{3^{2n}}$,

$$\alpha = \underbrace{\begin{pmatrix}(0,0)\\(1,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(2,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(2^2,0)\end{pmatrix}\cdots\begin{pmatrix}(0,0)\\(2^{n-1},0)\end{pmatrix}}_{A}\underbrace{\begin{pmatrix}(0,0)\\(2^n,0)\end{pmatrix}\cdots\begin{pmatrix}(0,0)\\(2^{2n-1},0)\end{pmatrix}}_{B}\cdot$$
$$\underbrace{\begin{pmatrix}(0,0)\\(0,1)\\(0,2)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,3)\\(0,2\cdot 3)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,3^2)\\(0,2\cdot 3^2)\end{pmatrix}\cdots\begin{pmatrix}(0,0)\\(0,3^{n-1})\\(0,2\cdot 3^{n-1})\end{pmatrix}}_{C}\cdot$$
$$\underbrace{\begin{pmatrix}(0,0)\\(0,3^n)\\(0,2\cdot 3^n)\end{pmatrix}\cdots\begin{pmatrix}(0,0)\\(0,3^{2n-1})\\(0,2\cdot 3^{2n-1})\end{pmatrix}}_{D},$$

and $g \in G$. Clearly, $\alpha \in \Lambda(G)$. The sums of the blocks in B and D are subgroups of G. For each block in A randomly pick an element except $(0,0)$ from the subgroup generated by the blocks in D and add the element onto the non-identity element in the block of A (these additions are selective shifts, see Section 5.1.9). For each block in C randomly pick two distinct non-identity elements from the subgroup generated by the blocks in B and add them onto the two non-identity elements in the block of C. Call the resulting logarithmic signature α'. We have $\ell(\alpha') = \ell(\alpha) = 2n + 2n + 3n + 3n = 10n$.

For example, for $n = 3$ we could get

$$\alpha' = \begin{pmatrix}(0,0)\\(1,2\cdot 3^3+3^4+2\cdot 3^5)\end{pmatrix}\begin{pmatrix}(0,0)\\(2,3^4+2\cdot 3^5)\end{pmatrix}\begin{pmatrix}(0,0)\\(2^2,2\cdot 3^3+2\cdot 3^4)\end{pmatrix}\begin{pmatrix}(0,0)\\(2^3,0)\end{pmatrix}\cdot$$
$$\begin{pmatrix}(0,0)\\(2^4,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(2^5,0)\end{pmatrix}\begin{pmatrix}(0,0)\\(2^3+2^4+2^5,1)\\(2^3+2^5,2)\end{pmatrix}\begin{pmatrix}(0,0)\\(2^4+2^5,3)\\(2^3,6)\end{pmatrix}\begin{pmatrix}(0,0)\\(2^4,3^2)\\(2^4+2^5,2\cdot 3^2)\end{pmatrix}\cdot$$
$$\begin{pmatrix}(0,0)\\(0,3^3)\\(0,2\cdot 3^3)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,3^4)\\(0,2\cdot 3^4)\end{pmatrix}\begin{pmatrix}(0,0)\\(0,3^5)\\(0,2\cdot 3^5)\end{pmatrix} \in \Lambda(\mathbb{Z}_{2^{2\cdot 3}} \oplus \mathbb{Z}_{3^{2\cdot 3}}).$$

In order to determine the run-time of Algorithm 8.32 for logarithmic signatures α' as constructed above, we regard the components of G separately.

When only looking at the first component of G, there are $3n$ blocks (namely the ones in A, B and C) containing non-identity elements. We have $|L'(\alpha', g, 1)| = 3^n$.

Analogously, when only looking at the second component of G, there are $3n$ blocks (namely the ones in A, C and D) containing non-identity elements. We have $|L'(\alpha', g, 2)| = 2^n$.

Both 3^n and 2^n are exponential in $\ell(\alpha')$. Thus the factorization Algorithm 8.32 has an exponential run-time for α'.

8.8.2. Dynamically Chosen Factor Groups

We generalize the idea from the previous section. φ_c projected group elements onto their cth component and factorizations with respect to this component only were regarded. This is equivalent to moving into the factor group $G/\langle e_1, e_2, \ldots, e_{c-1}, e_{c+1}, \ldots, e_m\rangle$ (where $e_i = (\delta_{i,1}, \delta_{i,2}, \ldots, \delta_{i,m}) \in G$ with $\delta_{i,j}$ the Kronecker delta) and regarding factorizations there.

The main problem of Algorithm 8.32 is that it regarded factorizations in statically chosen factor groups. As seen in the high run-time examples, it can happen that there are many solutions in every factor group. One improvement is to choose the factor groups dynamically (based on α), in the hope that there are factor groups with few solutions.

Let $N_1, N_2, \ldots, N_k \trianglelefteq G$ with $N_1 + N_2 + \ldots + N_k = G$ and $|N_1| \cdot |N_2| \cdots |N_k| = |G|$. Define

$$\overline{N_c} := N_1 + N_2 + \ldots + N_{c-1} + N_{c+1} + \ldots + N_k,$$

$\varphi_c : G \to G/\overline{N_c} : x \mapsto x + \overline{N_c}$ and $\alpha_c := \varphi_c(\alpha)$. Then

$$\begin{aligned} &\alpha[1][i_1] + \alpha[2][i_2] + \ldots + \alpha[n][i_n] = g \\ \Leftrightarrow\ &\varphi_c(\alpha[1][i_1] + \alpha[2][i_2] + \ldots + \alpha[n][i_n]) = \varphi_c(g) \text{ for all } 1 \leq c \leq k \\ \Leftrightarrow\ &\alpha_c[1][i_1] + \alpha_c[2][i_2] + \ldots + \alpha_c[n][i_n] = \varphi_c(g) \text{ for all } 1 \leq c \leq k. \end{aligned}$$

Algorithm 8.32 can be modified in a straightforward way to regard the N_c instead of the static components. Let us call the modified version Algorithm 8.32$^\diamond$.

The run-time of Algorithm 8.32$^\diamond$ is dependent on how the N_c are chosen. If they can be chosen in such a way that there are few solutions in each $L'(\alpha, g, N_c)$ (defined analogously to $L'(\alpha, g, c)$), the algorithm can be efficient.

One approach for choosing the N_c is to take the cyclic subgroups generated by the elements in $E(\alpha)$, removing as many subgroups as possible (e.g. a subgroup might lie fully within another subgroup and thus can be removed) and ensuring that they are a direct sum for G.

For the Example 8.34, this results in an efficient factorization algorithm. Every N_c would contain $(0, \ldots, 0)$ and one of the basis vectors (columns from the matrix). In every $G/\overline{N_c}$, there is exactly one block containing a non-$(0, \ldots, 0)$ element, i.e. $|L'(\alpha, g, N_c)| = 1$ for all $1 \leq c \leq n$.

However, the algorithm does not provide any way to compute factorizations when G is near a cyclic group. Algorithms 8.32 and 8.32$^\diamond$ required the existence of another algorithm that factors $\varphi_c(g)$. When G is cyclic of prime power order, there is only one $N_1 = G$ and the factorization problem is redirected to the other algorithm. In other words, in this case Algorithms 8.32 and 8.32$^\diamond$ do not result in any simplification and thus are useless.

8.8.3. Small Factor Groups

Let $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$, $\alpha = (A_1, A_2, \ldots, A_n)$ a canonical block sequence and $g \in G$.

Let $U \leq G$, $\varphi_U : G \to G/U : x \mapsto x + U$ and $\alpha' := \varphi_U(\alpha)$. If all blocks in α' except one block A'_i contain only $(0, \ldots, 0)$ elements (in G/U) and $\varphi_U(g)$ occurs only once in A'_i, the element to be selected in A'_i for a factorization of $\varphi_U(g)$ with respect to α' is determined, and thus the element to be selected in A_i for a factorization of g with respect to α is determined, too. Let $a \in A_i$ be the element that needs to be selected. Let α'' be the block sequence obtained by removing block A_i from α. The above procedure can now be repeated recursively for α'' instead of α and $g - a$ instead of g until all factorization indices are determined.

How do we choose the subgroup U? Clearly, most elements of $E(\alpha)$ should become $(0, \ldots, 0)$ when moving into the factor group G/U. Thus it is a good idea to construct U based on $E(\alpha)$. One way is the following. Build a list L containing all elements from $E(\alpha) \setminus \{(0, \ldots, 0)\}$ sorted ascendingly by their orders. Set $U \leftarrow \{(0, \ldots, 0)\}$. For each h in L (from left to right), test whether $U + \langle h \rangle \neq G$ and if so, set $U \leftarrow U + \langle h \rangle$.

Observe the similarity to the generic factorization Algorithm 8.30. Algorithm 8.30 chooses the normal subgroups in a more complex way (estimating the complexity of the block sequence in the factor group, etc.), but they are also built based on subgroups generated by single elements from the logarithmic signature and the generator element's order has a high influence.

Unfortunately, the U constructed by the procedure above does not necessarily result in an α' with only one block containing non-$(0, \ldots, 0)$ elements.

When there are multiple candidates in one block, the decision which of them is the correct one can only be made in a larger factor group. It can happen that the decision is only possible in the full G, not in any proper factor group of it.

Example 8.36. Let $n \in \mathbb{N}$ even, $G = \mathbb{Z}_{2^n}$ and $\beta = (B_1, B_2, \ldots, B_n) \in \Lambda(G)$ with $B_i = (0, 2^{i-1})$. Define $\alpha = (A_1, A_2, \ldots, A_{\frac{n}{2}})$ with $A_i := B_i + B_{n-i+1} = \{a + b \mid a \in B_i, b \in B_{n-i+1}\}$ for all $1 \leq i \leq \frac{n}{2}$. Clearly, $\alpha \in \Lambda(G)$.

For example, for $n = 6$ we get

$$\beta = \begin{pmatrix}0\\1\end{pmatrix}\begin{pmatrix}0\\2\end{pmatrix}\begin{pmatrix}0\\4\end{pmatrix}\begin{pmatrix}0\\8\end{pmatrix}\begin{pmatrix}0\\16\end{pmatrix}\begin{pmatrix}0\\32\end{pmatrix} \rightsquigarrow \alpha = \begin{pmatrix}0\\1\\32\\33\end{pmatrix}\begin{pmatrix}0\\2\\16\\18\end{pmatrix}\begin{pmatrix}0\\4\\8\\12\end{pmatrix}.$$

For all subgroups $U \leq G$ with $|U| \geq 2^{\frac{n}{2}}$, $\varphi_U(\alpha)$ does not determine any factorization index, i.e. there are at least two elements in every block of $\varphi_U(\alpha)$ that both are part of a factorization of $\varphi_U(g)$.

Block A_1 contains two elements of order 2^n. The decision which of them is required for a factorization of g with respect to α can only be made in the full G, not in any proper factor group of G (because in every proper factor group, the two elements of order 2^n are identical, as their difference is 2^{n-1}; $\{0, 2^{n-1}\}$ is the smallest non-trivial subgroup of G, and all other non-trivial subgroups are supersets of $\{0, 2^{n-1}\}$).

Algorithm 8.30 handles periodic blocks elegantly. In Example 8.36, on the deepest recursion level the decision whether a highest order element is part of a factorization of g is

made, and on the first recursion level it is decided which of the two highest order elements actually is correct.

In contrast, the new algorithm cannot handle periodic blocks in a satisfying way yet. In order to make the decision which of the two highest order elements has to be selected, the algorithm needs to move into a larger factor group. The larger factor group should again have the property that most elements in the logarithmic signature should be the identity. Algorithm 8.30 constructs such an optimal subgroup chain, but from top to bottom. However, when using exactly this chain, there does not seem to be any advantage of the new algorithm over Algorithm 8.30; in fact, the new algorithm would just be the second phase (moving up from the recursion) of Algorithm 8.30. When not using this chain, another one would need to be constructed from bottom up.

8.9. Tame Logarithmic Signatures

8.9.1. Amalgamated Transversal Logarithmic Signatures $\mathcal{AT}(G)$

Theorem 8.37. *Let G be an abelian group (represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$) and $\alpha \in \mathcal{AT}(G)$. Then α is tame.*

Proof. We can assume that α is canonical (see Proposition 7.1). We show that Algorithm 8.23 finds a periodic or subgroup block on every recursion level.

For the transformations applied by the amalgamated logarithmic signature generation algorithm to an exact transversal logarithmic signature of G, see Section 6.3.

We can ignore the first two transformations (permuting the blocks and elements within blocks), because Algorithm 8.23 does not care about the order of blocks or elements.

The generation algorithm started with an exact transversal logarithmic signature having a subgroup chain structure. Amalgamating two blocks (the fourth transformation) always results in a periodic block on the recursion level where one of the blocks would be a subgroup. Let $\alpha = (A_1, A_2, \ldots, A_r) \in \Lambda(G)$ with $A_1 \leq G$, $A_1 + A_2 \leq G$, etc., then amalgamating two blocks A_i, A_j with $i < j$ results in a periodic block (for which $A_i \setminus \{(0, \ldots, 0)\}$ are periods, and possibly there are even more, e.g. in case A_j is a subgroup in $G/(A_1 + \ldots + A_i)$), which is found and decomposed into A_i and A_j again on recursion level i. Note that if $A_i + A_j$ even is a subgroup in $G/(A_1 + \ldots + A_{i-1})$, then the algorithm on recursion level i directly moves into the corresponding factor group, without decomposing the periodic block $A_i + A_j$ first.

Lastly, observe that translations (the third transformation) do not interfere with the above. If two blocks A_i, A_j (without any translations) are amalgamated to a new block $A := A_i + A_j$ with period $g \in G \setminus \{(0, \ldots, 0)\}$, then $A' := A_i + t + A_j + s$ also has period g for arbitrary $t, s \in G$, because $(A_i + t + A_j + s) + g = (A_i + A_j + g) + t + s = (A_i + A_j) + t + s = A_i + t + A_j + s$. Also, translated subgroups are restored by the initial normalization: if A_i is a subgroup in $G/(A_1 + \ldots + A_{i-1})$, the normalization of a translation $A_i + t$ (for an arbitrary $t \in G$) results in A_i again, because normalization adds $-t - a$ for an arbitrary $a \in A_i$ onto $A_i + t$ (the elements could be permuted) and we

have $A_i + t - t - a = A_i - a = A_i$, i.e. the original A_i (within $G/(A_1 + \ldots + A_{i-1})$) is restored. □

8.9.2. Aperiodic Decomposition and Reunion

Proposition 8.38. *Let $G = \mathbb{Z}_2^n$ and $\beta \in \Lambda(G)$ generated by Algorithm 6.6 (in Section 6.4.2).*

Then β is tame.

Proof. Let $\beta = (\beta', B_2, \ldots, B_s) \in \Lambda(G)$ generated by Algorithm 6.6, and $\varphi_H : G \to G/H : g \mapsto g + H$ for $H \leq G$.

β' generates the subgroup $U_1 \oplus D_1$. Define $\beta^\diamond := (\varphi_{U_1 \oplus D_1}(B_2), \ldots, \varphi_{U_1 \oplus D_1}(B_s))$. Then $\beta^\diamond \in \Lambda(G/(U_1 \oplus D_1))$. Observe that $K_2 \subseteq U_1 \setminus \{(0, \ldots, 0)\}$. Thus $E(\varphi_{U_1 \oplus D_1}(K_2)) = \{(0, \ldots, 0)\}$ and therefore $B_2 = U_2 \oplus D_2$ in $G/(U_1 \oplus D_1)$. This can be iterated, i.e. every B_i is a subgroup in $G/(U_1 \oplus \ldots \oplus U_{i-1} \oplus D_1 \oplus \ldots \oplus D_{i-1})$. Consequently, $\beta^\diamond$ is tame by Algorithm 8.23.

The remaining factorization in β' can be found using an exhaustive search.

If the number of blocks in β' is unknown, simply try all possible block counts (increasing from 1; observing whether it results in a chain of factor groups that we can descend into); this still is efficient, because β' is small (constant size for $n \to \infty$). □

Theorem 8.39. *Let $G = \mathbb{Z}_2^n$ and $\alpha = (A_1, \ldots, A_m) \in \Lambda(G)$ generated by Algorithm 6.7 (in Section 6.4.2).*

Then α is tame.

Proof. All we need to show is that we can efficiently find a small subgroup T (with $|T|$ being polynomial in $\ell(\alpha)$) with $U_1 \oplus D_1 \subseteq T$. The rest then follows with Proposition 8.38 (move into the factor group G/T, etc.).

We analyze the transformations that Algorithm 6.7 has applied to the aperiodic logarithmic signature produced by Algorithm 6.6.

Element shuffles and block shuffles can be ignored; the following does not depend on the order of the blocks or elements within the blocks.

Observe the following properties of translations and fusions/amalgamations for all $B, C \subseteq G$ and $s, t \in G$:

- $(B + s) + t = B + (s + t)$, i.e. multiple translations are equivalent to one translation.
- $(B + s) + C = (B + C) + s$, i.e. translations and fusions can be interchanged.

Consequently, multiple translations and fusions of blocks are equivalent to fusing the blocks and applying exactly one translation (where the translation element is the sum of all single translations).

Furthermore, we need the following observation. Let $\pi : \mathcal{P}(G) \to \mathcal{P}(G) : M \mapsto \{g \in G \mid M + g = M\}$ compute the periods of M together with $(0, \ldots, 0)$ (this forms a subgroup, see Lemma 2.7). Let $H \leq G$, $S \subseteq G$, $t \in G$, and $M := H + S + t$. Clearly, $\pi(M) \supseteq H$.

Let $M' := M - y$ for any $y \in M$, then still $\pi(M') \supseteq H$. As we now additionally have $(0, \ldots, 0) \in M'$ (and thus all periods of M' are elements of M'), we get $H \subseteq M'$.

So, if we have a block M that actually is of the form $M = (U_1 \oplus D_1) + S + t$, we can perform a normalization and look at the periods to efficiently recover $U_1 \oplus D_1$ or a subgroup containing $U_1 \oplus D_1$.

Assume that the blocks generating $U_1 \oplus D_1$ were transformed to c blocks (if c is unknown, try all, like in the proof of Proposition 8.38).

For all $1 \leq i_1 < \ldots < i_c \leq m$ do the following:

1. Compute the sum set $M \leftarrow A_{i_1} + \ldots + A_{i_c}$.
2. Compute $M' \leftarrow M - y$ for an arbitrary $y \in M$.
3. Compute $T \leftarrow \pi(M')$.
4. Test whether moving into G/T results in a logarithmic signature that allows iteratively moving into factor groups (by subgroups of periods). If so: move into G/T, apply Algorithm 8.23, and use an exhaustive search in the remaining blocks to efficiently find the factorization of any group element. If not: try the next tuple $(i_1, \ldots, i_c)$.

This procedure in total has a run-time polynomial in $\ell(\alpha)$, because:

- There are $\binom{m}{c}$ possibilities for the tuples $(i_1, \ldots, i_c)$, and $\binom{m}{c} \leq m^c$; this is polynomial in $m \leq \ell(\alpha)$, if c is constant (which we assume).
- $|M| = \prod_{i \in \{i_1, \ldots, i_c\}} |A_i| \leq \ell(\alpha)^c$, i.e. $|M|$ is polynomial in $\ell(\alpha)$.
- As $(0, \ldots, 0) \in M'$, we have $T \subseteq M'$. Thus T can be computed efficiently (test for each element in M' whether it is a period for M'; this is possible in time polynomial in $|M'|$, and $|M'| = |M| \leq \ell(\alpha)^c$).
- It is clear that testing the logarithmic signature for the iterative property is possible in time polynomial in $\ell(\alpha)$; see the factorization algorithms that work like this.
- As $|M| \leq \ell(\alpha)^c$, the exhaustive search in the remaining blocks is possible efficiently.

□

8.9.3. Strongly Aperiodic Logarithmic Signatures

Proposition 8.40. *Let $G = \mathbb{Z}_p^{3s}$ and $\beta \in \Lambda(G)$ generated by Algorithm 6.8 (in Section 6.4.3); β is strongly aperiodic and $t(\beta) = (p^3, \ldots, p^3)$.*

Then β is tame (i.e. factorizations can be computed in time polynomial in $\mathcal{S}(G, \beta) = \lceil \log_2 |G| \rceil \cdot \ell(\beta) = \lceil \log_2 p^{3s} \rceil \cdot s \cdot p^3$; both parameters may grow).

Proof. Let $\beta = (B_1, \ldots, B_s)$. Let us assume that we know the vectors v_1 and v_2 (more on this at the end of the proof). Write $\varphi_H : G \to G/H : g \mapsto g + H$ for $H \leq G$.

For all $2 \leq i \leq s$ we have:

$$\begin{aligned} D_i &= \{(0, \ldots, 0), v_{2s+i}, 2v_{2s+i}, \ldots, (p-1)v_{2s+i}\}, \\ A_i^{(j)} &= \langle v_{2i-1} + (j-1)v_1,\ v_{2i} + (j-1)v_2 \rangle \text{ for } j \in \{1, \ldots, p\}, \\ B_i &= \bigcup_{d_{i,j} \in D_i} (d_{i,j} + A_i^{(j)}). \end{aligned}$$

When moving into the factor group $G/\langle v_1, v_2 \rangle$, the following happens:

$$\begin{aligned} \varphi_{\langle v_1,v_2\rangle}(B_i) &= \varphi_{\langle v_1,v_2\rangle}\Big(\bigcup_{d_{i,j} \in D_i} (d_{i,j} + A_i^{(j)})\Big) \\ &= \bigcup_{d_{i,j} \in D_i} (\varphi_{\langle v_1,v_2\rangle}(d_{i,j}) + \varphi_{\langle v_1,v_2\rangle}(A_i^{(j)})) \\ &= \bigcup_{d_{i,j} \in D_i} (\varphi_{\langle v_1,v_2\rangle}(d_{i,j}) + \varphi_{\langle v_1,v_2\rangle}(\langle v_{2i-1}, v_{2i}\rangle)) \\ &= \varphi_{\langle v_1,v_2\rangle}\left(D_i + \langle v_{2i-1}, v_{2i}\rangle\right). \end{aligned}$$

Thus, when moving into the factor group $G/\langle v_1, v_2 \rangle$, all blocks B_i with $2 \leq i \leq s$ become subgroups, because D_i is a (cyclic) subgroup and $\langle v_{2i-1}, v_{2i}\rangle$ also is a subgroup (and the sum of two subgroups is a subgroup, as G is abelian).

Consequently, $(B_2, \ldots, B_s)$ in $G/\langle v_1, v_2 \rangle$ is tame by Algorithm 8.23. In order to factor a $g \in G$, for each $x \in B_1$ try to factor $\varphi_{\langle v_1,v_2\rangle}(g - x)$ with respect to $(B_2, \ldots, B_s)$ in $G/\langle v_1, v_2 \rangle$ and test the combined factorization indices with respect to β in G. So, β is tame.

Above we assumed that v_1 and v_2 are known, thus it remains to show that this assumption is not a problem. Due to $(0, \ldots, 0) \in D_1$ and $A_1^{(1)} = \langle v_1, v_2 \rangle$, we have $v_1, v_2 \in B_1$. Therefore we can simply try all $\binom{p^3-1}{2} = \frac{(p^3-1)(p^3-2)}{2}$ possibilities for v_1, v_2 (for each of these candidate pairs $\{u, v\} \subseteq (E(B_1) \setminus \{(0, \ldots, 0)\})$ with $u \neq v$ check whether all other blocks become subgroups when moving into the factor group $G/\langle u, v \rangle$ and if so continue with the factorization process as above). □

Proposition 8.41. *Let $G = \mathbb{Z}_2^{2s-1}$ and $\beta \in \Lambda(G)$ generated by the algorithm in Section 8 of [Sta13].*

Then β is tame.

Proof. The attack idea from the proof of Proposition 8.40 can be used.

Note that there is a minor difference, which however does not affect the attack. In Proposition 8.40, all blocks except B_1 become subgroups at once when moving into $G/\langle v_1, v_2 \rangle$. However, here in Proposition 8.41, only one block of β is guaranteed to become a subgroup: when moving into $G/\langle v_1, v_2 \rangle$, block B_2 becomes a subgroup; when moving into $G/(\langle v_1, v_2 \rangle + B_2)$, block B_3 becomes a subgroup; when moving into $G/(\langle v_1, v_2 \rangle + B_2 + B_3)$,

block B_4 becomes a subgroup; and so on. As the factorization algorithm moves into factor groups iteratively anyway, this does not affect the attack. □

8.9.4. Specific Group and Logarithmic Signature Types

Lemma 8.42. *Let G be an abelian group and $\alpha = (A_1, \ldots, A_n) \in \Lambda(G)$ canonical of type $t(\alpha) = (q_1, \ldots, q_n)$. Let $p, q, r, s \in \mathbb{P}$ distinct, $\lambda, \mu \in \mathbb{N}$ and $3 \leq a \in \mathbb{N}$.*

In all of the following cases, α has a subgroup block:

- $t(\alpha) \in \mathbb{P}^n$.
- $t(G) = (2, \ldots, 2)$, $q_1 = 2$.
- $t(G) = (2, \ldots, 2)$, $t(\alpha) = (4, \ldots, 4)$.
- $t(G) = (2, \ldots, 2, 2p)$, $t(\alpha) = (2, \ldots, 2, 2p)$.
- *G is a group or a subgroup of a group of type*

$$(p^2), (p, p), (p, q), (4, 2), (2, 2, 2, 2).$$

In all of the following cases, α has a periodic block:

- *The p-component of G is cyclic and $q_1, \ldots, q_{n-1} \in \{p^i \mid i \in \mathbb{N}\}$.*
- *G is cyclic and $q_1, \ldots, q_n \in \big(\{z^i \mid z \in \mathbb{P}, i \in \mathbb{N}\} \cup \{yz \mid y, z \in \mathbb{P}\}\big)$.*
- $t(G) = (2^\lambda, 2, \ldots, 2)$, $q_1, \ldots, q_n \in \{2, 4\}$.
- $t(G) = (2^\lambda, 2^\mu)$, $q_1, \ldots, q_n \in \{2, 4\}$.
- $t(G) = (2^\lambda, 2, \ldots, 2)$, $q_1 = \ldots = q_{n-1} = 2$.
- $q_1 = 4$, $q_2, \ldots, q_n \in \mathbb{P}$.
- *The 2-component of G is elementary, $q_1, \ldots, q_n \in (\mathbb{P} \cup \{4\})$.*
- *G is a group or a subgroup of a group of type*

$$(p^a, q), (p^2, q^2), (p^2, q, r), (p, q, r, s), (p^3, 2, 2), (p^2, 2, 2, 2), (p, 2^2, 2),$$
$$(p, 2, 2, 2, 2), (p, q, 2, 2), (p, 3, 3), (3^2, 3), (2^a, 2), (2^2, 2^2), (p, p).$$

Proof. In the first case there is a subgroup block by Rédei's theorem (Theorem 8.1). In the other cases the existence of a subgroup block is shown in [Sza04] (in the same order as above): Theorem 4.1.1 (three cases), Section 4.1.

The periodicity is shown in [Sza04] (in the same order as above): Theorem 6.1.3, Theorem 6.2.1, Theorem 7.2.2, Theorem 7.2.4, Corollary 7.2.1, Theorem 7.4.1, Theorem 7.4.2, Chapter 8. □

Observation 8.43. In all of the cases of Lemma 8.42, Algorithm 8.23 descends at least once in the recursion, i.e. factoring in the starting logarithmic signature is as hard as factoring in the logarithmic signature for the factor group G/N, where N is the subgroup block or the subgroup block resulting from splitting the periodic block.

In some cases, this idea can be iterated recursively. If the logarithmic signature for the factor group G/N is again of one of the forms in Lemma 8.42 (i.e. it again has a periodic block), the algorithm can descend at least once more. If this can be iterated recursively until reaching the $\{0\}$ group, the starting logarithmic signature clearly is tame.

We would like to state a summary when such full recursions occur. For this, we need to know how the type of the logarithmic signature and the type of the group change when moving into a factor group.

Let $\alpha = (A_1, A_2, \ldots, A_n) \in \Lambda(G)$ contain a periodic block A_i, let N be the periods of A_i together with 0 (we have $N \leq G$ by Lemma 2.7), and $\varphi : G \to G/N : x \mapsto x + N$. Define

$$\beta := (E(\varphi(A_1)), E(\varphi(A_2)), \ldots, E(\varphi(A_n))).$$

We claim that $\beta \in \Lambda(G/N)$. In order to see why this is true, first have a look at $\varphi(\alpha)$. $\varphi(\alpha)$ is an $|N|$-factorization of G/N (by Theorem 7.3). As the block A_i is periodic with periods N, in G/N the application of E reduces the size of the block $\varphi(A_i)$ by the factor $|N|$ (proof: when splitting $A_i = B + N$ for a $B \subseteq G$ using Lemma 2.7, N becomes $\{0\}$ in G/N), i.e. $|E(\varphi(A_i))| = \frac{|A_i|}{|N|}$. So, the multiplicity of the factorization when moving from α (for G) to $\varphi(\alpha)$ (for G/N) is generated only by block A_i. In the definition of β this multiplicity is removed by applying E. Consequently $\beta \in \Lambda(G/N)$.

Thus, for all $j \neq i$, we have $|E(\varphi(A_j))| = |A_j|$. Write $t(\alpha) = (u_1, \ldots, u_n)$ and $t(\beta) = (v_1, \ldots, v_n)$. Then

$$v_1 \mid u_1,\ v_2 \mid u_2,\ \ldots,\ v_n \mid u_n.$$

The factorization algorithm removes any blocks becoming $\{0\}$, i.e. any "1"s are removed from the $t(\beta)$ vector.

In summary, when the factorization algorithm moves into a factor group, each value in the type vector of the logarithmic signature either does not change, vanishes completely or is reduced to one of its proper divisors (except 1).

It remains to understand how the group type vector changes when moving into a factor group. For this, we use a well-known result on subgroups of a group:

Lemma 8.44. *Let G be an abelian group (represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$), and $U \leq G$. Then there exist $0 \leq k_i' \leq k_i$ for all $1 \leq i \leq m$ such that*

$$U \cong \mathbb{Z}_{p_1^{k_1'}} \oplus \mathbb{Z}_{p_2^{k_2'}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m'}}.$$

Proof. This is proven in Theorem 10.7 of [Fuc67]. □

Furthermore, it is well known that every factor group of an abelian group G is isomorphic to a subgroup of G (and the other way around).

So, when the factorization algorithm moves into a factor group, each value in the type vector of the group either does not change, vanishes completely or is reduced to one of its proper divisors (except 1; $t(G/N)$ shall not contain any 1).

Note that we here just need this statement about the group type vector; we do not need to convert elements to the factor group representation. Algorithm 8.23 builds a chain of subgroups for computing in factor groups.

Having understood how the type vectors change, we can now create a list of conditions that result in the factorization algorithm finding a periodic block on every recursion level (and thus the logarithmic signature being tame). We only list the cases describing infinite families.

Theorem 8.45. *Let G be an abelian group (represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$), and $\alpha = (A_1, \ldots, A_n) \in \Lambda(G)$ canonical of type $t(\alpha) = (q_1, \ldots, q_n)$. Let $p, q \in \mathbb{P}$ distinct, $\lambda, \mu \in \mathbb{N}$ and $3 \leq a \in \mathbb{N}$.*

In all of the following cases, α is tame (in the families of logarithmic signatures, all parameters may tend towards infinity):

1. $t(\alpha) \in \mathbb{P}^n$.
2. *The p-component of G is cyclic and* $q_1, \ldots, q_{n-1} \in \{p^i \mid i \in \mathbb{N}\}$.
3. G *is cyclic and* $q_1, \ldots, q_n \in \left(\{z^i \mid z \in \mathbb{P}, i \in \mathbb{N}\} \cup \{yz \mid y, z \in \mathbb{P}\}\right)$.
4. $t(G) = (2^\lambda, 2, \ldots, 2)$, $q_1, \ldots, q_n \in \{2, 4\}$.
5. $t(G) = (2^\lambda, 2^\mu)$, $q_1, \ldots, q_n \in \{2, 4\}$.
6. $t(G) = (2^\lambda, 2, \ldots, 2)$, $q_1 = \ldots = q_{n-1} = 2$.
7. $q_1 = 4$, $q_2, \ldots, q_n \in \mathbb{P}$.
8. *The 2-component of G is elementary,* $q_1, \ldots, q_n \in (\mathbb{P} \cup \{4\})$.
9. G *is a group or a subgroup of a group of type* (p^a, q) *or* $(2^a, 2)$.

Proof. By Observation 8.43, it is sufficient to show that when the factorization algorithm moves into any factor group, one of the conditions is fulfilled again for the factor group and its logarithmic signature. Let $G/N \neq \{0\}$ be the factor group and $\beta \in \Lambda(G/N)$ the canonical logarithmic signature of type $t(\beta) = (u_1, \ldots, u_s)$ that the factorization algorithm produces for G/N.

1. $t(\beta) \in \mathbb{P}^s$ (same case, i.e. β has the same form as α and thus again contains a periodic block, and this can be iterated recursively).
2. The p-component of G/N is cyclic and $u_1, \ldots, u_{s-1} \in \{p^i \mid i \in \mathbb{N}\}$ (same case; the last block is irrelevant for the tameness).

3. Factor groups of cyclic groups are cyclic, and again
$$u_1, \ldots, u_s \in \left(\{z^i \mid z \in \mathbb{P}, i \in \mathbb{N}\} \cup \{yz \mid y, z \in \mathbb{P}\}\right)$$
(same case).

4. $t(G/N) = (2^{\lambda'}, 2, \ldots, 2)$ for some $1 \leq \lambda' \leq \lambda$, and $u_1, \ldots, u_s \in \{2, 4\}$ (same case).

5. $u_1, \ldots, u_s \in \{2, 4\}$. Two cases can occur:
   - $t(G/N) = (2^{\lambda'}, 2^{\mu'})$ with $1 \leq \lambda' \leq \lambda$ and $1 \leq \mu' \leq \mu$ (same case).
   - $t(G/N) = (2^v)$ with $1 \leq v \leq \max\{\lambda, \mu\}$ (see the case for G/N cyclic and $u_1, \ldots, u_s \in \left(\{z^i \mid z \in \mathbb{P}, i \in \mathbb{N}\} \cup \{yz \mid y, z \in \mathbb{P}\}\right)$, with $y = z = 2$ we get size 4).

6. $t(G/N) = (2^{\lambda'}, 2, \ldots, 2)$ for some $1 \leq \lambda' \leq \lambda$, and $u_1 = \ldots = u_{s-1} = 2$ (same case; the last block is irrelevant for the tameness).

7. Two cases can occur:
   - $u_1 = 4$, $u_2, \ldots, u_s \in \mathbb{P}$ (same case).
   - $t(\beta) \in \mathbb{P}^s$.

8. $u_1, \ldots, u_s \in (\mathbb{P} \cup \{4\})$. Two cases can occur:
   - G/N has a 2-component. Then the 2-component of G/N is elementary (same case).
   - G/N does not have a 2-component. This implies $2 \nmid |G/N|$ and thus $t(\beta) \in \mathbb{P}^s$.

9. Let us assume $t(G) = (p^a, q)$ or $t(G) = (2^a, 2)$. If $t(G) = (p^a, q)$, let $u := p$ and $v := q$, otherwise let $u := 2$ and $v := 2$. Three cases can occur:
   - $t(G/N) = (u^{a'}, v)$ for some $1 \leq a' \leq a$ (same case).
   - $t(G/N) = (u^{a'})$ for some $1 \leq a' \leq a$. Then G/N is cyclic and $u_1, \ldots, u_s \in \left(\{z^i \mid z \in \mathbb{P}, i \in \mathbb{N}\} \cup \{yz \mid y, z \in \mathbb{P}\}\right)$.
   - $t(G/N) = (v)$. Then β contains only one block and thus is tame. □

Remark 8.46. Let $G = \mathbb{Z}_2^k$ and $\alpha = (A_1, \ldots, A_n) \in \Lambda(G)$ with $|A_i| \in \{2, 4\}$ for all $1 \leq i \leq n$, then α is tame (by Theorem 8.45, case 4 with $\lambda = 1$). This special case was also proven in [Nus11] (with a different approach using graphs and matrices).

8.10. Factoring by Solving an Integer Linear Programming Problem

Finding the factorization of an element with respect to a logarithmic signature of an abelian group can be formulated as an integer linear programming (ILP) problem. However, solving ILPs in general seems to be a much more difficult problem than factoring.

Let G be an abelian group of type $t(G) = (m_1, \ldots, m_c)$, $\alpha \in \Lambda(G)$ of type $t(\alpha) = (r_1, \ldots, r_s)$, and $g = (g_1, \ldots, g_c) \in G$ the element to be factored. Define two mappings $\rho : \{1, \ldots, s+1\} \to \mathbb{N}_0 : k \mapsto \sum_{i=1}^{k-1} r_i$ (counts the elements in all blocks left of the kth block) and $\gamma_k : G \to \mathbb{N}_0 : (h_1, \ldots, h_c) \mapsto h_k$ (projection of the kth component of a group element).

Define an ILP problem with $\ell(\alpha) + c$ integer variables $x_1, \ldots, x_{\ell(\alpha)+c}$ and the following constraints (in standard form):

- $x_1 \geq 0, x_2 \geq 0, \ldots, x_{\ell(\alpha)+c} \geq 0$.
- For all $1 \leq k \leq s$:
$$\sum_{i=\rho(k)+1}^{\rho(k+1)} x_i = 1.$$
- For all $1 \leq k \leq c$:
$$\sum_{i=1}^{s} \sum_{j=1}^{r_i} (x_{\rho(i)+j} \cdot \gamma_k(\alpha[i][j])) - x_{\ell(\alpha)+k} \cdot m_k = g_k.$$

This ILP problem has exactly one solution, thus the linear function to be maximized is irrelevant.

The first $\ell(\alpha)$ variables $x_1, \ldots, x_{\ell(\alpha)}$ are actually binary variables (i.e. must be either 0 or 1). They can be interpreted as coefficients of the elements in the blocks of α. The "$= 1$" constraints together with the non-negativity constraints ensure that exactly one element in each block has a 1 coefficient, all others must be 0. The last c constraints ensure that the sum of the selected elements give g. This is achieved component-wise: for each cyclic group component of G a constraint ensures that the sum of these components equals the component of the element to be factored. The "$-x_{\ell(\alpha)+k} \cdot m_k$" summand realizes a "mod m_k" computation.

Example 8.47. Let $G = \mathbb{Z}_2 \oplus \mathbb{Z}_3 \oplus \mathbb{Z}_3 \oplus \mathbb{Z}_5$, $g = (0, 1, 2, 1) \in G$ and

$$\alpha = \begin{pmatrix} (0,0,0,0) \\ (1,1,1,4) \\ (0,1,0,2) \\ (0,2,0,1) \\ (1,0,1,3) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (0,1,1,0) \\ (0,0,2,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,0,0,0) \end{pmatrix} \begin{pmatrix} (0,0,0,0) \\ (1,1,0,0) \\ (0,2,0,0) \end{pmatrix} \in \Lambda(G).$$

The ILP problem for factoring g with respect to α is

$$\begin{aligned} x_1 &\geq 0, \\ &\vdots \\ x_{17} &\geq 0, \\ x_1 + x_2 + x_3 + x_4 + x_5 &= 1, \end{aligned}$$

$$\begin{aligned} x_6 + x_7 + x_8 &= 1, \\ x_9 + x_{10} &= 1, \\ x_{11} + x_{12} + x_{13} &= 1, \\ x_2 + x_5 + x_{10} + x_{12} - x_{14} \cdot 2 &= 0, \\ x_2 + x_3 + x_4 \cdot 2 + x_7 + x_{12} + x_{13} \cdot 2 - x_{15} \cdot 3 &= 1, \\ x_2 + x_5 + x_7 + x_8 \cdot 2 - x_{16} \cdot 3 &= 2, \\ x_2 \cdot 4 + x_3 \cdot 2 + x_4 + x_5 \cdot 3 - x_{17} \cdot 5 &= 1. \end{aligned}$$

Solving this using a standard ILP solver, we obtain the solution $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 1, x_5 = 0, x_6 = 0, x_7 = 0, x_8 = 1, x_9 = 1, x_{10} = 0, x_{11} = 0, x_{12} = 0, x_{13} = 1, x_{14} = 0, x_{15} = 1, x_{16} = 0, x_{17} = 0$. Thus the factorization of g with respect to α is

$$(0,2,0,1) + (0,0,2,0) + (0,0,0,0) + (0,2,0,0) = (0,1,2,1) = g.$$

Implementation. We have implemented this factorization approach (method `DmMultiZnGroup.FactorByILP`), using the well-known lp_solve[1] library.

Furthermore, we implemented a function that creates a *.lp* ILP problem file for a given factorization problem, which can then be solved for instance using the Gurobi[2] ILP solver.

Run-time. In general, solving ILPs is known to be NP-hard. We have experimented with factoring using this approach, however the performance was not acceptable. Although both lp_solve and Gurobi implement various solver strategies and optimizations, even for only medium-sized logarithmic signatures no solutions were found in a reasonable amount of time. If one wants to use this approach for factoring, solver algorithms would need to be optimized strongly for this special type of problem. Without optimizations, this approach is impractical.

For example, let $G = \mathbb{Z}_2^n$, $g = (1,0,1,0,\ldots) \in G$, and $\alpha \in \Lambda(G)$ obtained by the simple generation procedure below. This procedure starts with a simple logarithmic signature for G and performs selective shifts, where some of the indices of the involved blocks are generated using the Advanced Encryption Standard (AES) block cipher in counter (CTR) mode.

The generated logarithmic signature looks random; there do not seem to be any special properties that the ILP solver could exploit.

Although we could have used LS-Gen from Section 6.5 for generating α, we chose to use the following simple generation procedure for easier reproducibility of the results.

1. Set $\alpha \leftarrow \begin{pmatrix}(0,\ldots,0)\\(1,0,0,\ldots)\end{pmatrix}\begin{pmatrix}(0,\ldots,0)\\(0,1,0,0,\ldots)\end{pmatrix}\begin{pmatrix}(0,\ldots,0)\\(0,0,1,0,0,\ldots)\end{pmatrix}\cdots\begin{pmatrix}(0,\ldots,0)\\(0,\ldots,0,1)\end{pmatrix} \in \Lambda(G)$.

[1] http://lpsolve.sourceforge.net/
[2] http://www.gurobi.com/

2. Set $K \leftarrow (0, \ldots, 0) \in \mathbb{Z}_2^{256}$ (constant key for AES-256).
3. For $i \leftarrow 0$ to $n^2 - 1$:
   a) Set $x \leftarrow i \bmod n$.
   b) Set $v \leftarrow \text{AES-256}_K(i)$ (where i is converted to an array of 16 bytes, little-endian).
   c) Set $y \leftarrow v \bmod n$ (where v is converted to a 128-bit unsigned integer, little-endian).
   d) If $x \neq y$:
      - Set $\alpha[x+1][2] \leftarrow \alpha[x+1][2] \oplus \alpha[y+1][2]$.

We have implemented this generation procedure in the LogSig utility (Chapter 12). With the command line option "`-BenchmarkILP`", on a desktop PC (AMD A6-3650 2.60 GHz, 8 GB RAM, Windows 7 SP1 64-bit) we obtained the lp_solve run-times in the table below (without counting the time required for building α and g). With the command line option "`-CreateLPFiles`", LogSig outputs *.lp* files for the factorization problems; we measured the time that a `gurobi_cl` invocation (with the *.lp* file path as parameter) required to find a solution.

Note that all logarithmic signatures of type $(2, \ldots, 2)$ for $G = \mathbb{Z}_2^n$ are actually tame (e.g. see Algorithm 8.23). For comparison, we also list the run-times of our implementations of Algorithm 8.23 and Algorithm 8.30.

n	lp_solve 5.5.2	Gurobi 5.6.3	Alg. 8.23	Alg. 8.30
8	0.005 s	0.023 s	$<$ 0.001 s	0.018 s
9	0.006 s	0.038 s	$<$ 0.001 s	0.027 s
10	0.006 s	0.040 s	$<$ 0.001 s	0.055 s
11	0.007 s	0.030 s	$<$ 0.001 s	0.077 s
12	0.012 s	0.038 s	0.001 s	0.119 s
13	0.004 s	0.051 s	0.001 s	0.158 s
14	0.053 s	0.120 s	0.002 s	0.259 s
15	0.100 s	0.025 s	0.002 s	0.359 s
16	0.045 s	0.102 s	0.003 s	0.442 s
17	0.136 s	0.215 s	0.004 s	0.633 s
18	0.187 s	0.229 s	0.004 s	0.780 s
19	0.827 s	0.615 s	0.006 s	1.030 s
20	2.637 s	0.584 s	0.007 s	1.313 s
21	1.844 s	1.946 s	0.007 s	1.616 s
22	0.302 s	5.066 s	0.010 s	2.246 s
23	2.806 s	1.874 s	0.012 s	2.803 s
24	2.029 s	29.289 s	0.012 s	3.285 s
25	39.887 s	19.120 s	0.014 s	3.829 s
26	149.802 s	61.606 s	0.018 s	5.053 s
27	115.328 s	108.663 s	0.019 s	6.107 s

n	lp_solve 5.5.2	Gurobi 5.6.3	Alg. 8.23	Alg. 8.30
28	445.739 s	359.582 s	0.022 s	6.957 s
29	1367.553 s	256.763 s	0.025 s	8.686 s
30	867.829 s	252.540 s	0.028 s	10.158 s
31	2089.052 s	1897.211 s	0.033 s	12.392 s

Table 8.1.: Run-Times for Factoring by Integer Linear Programming Problem Solving

8.11. Counting Logarithmic Signatures

In this section we show for a few groups that the number of logarithmic signatures (i.e. keys for PGM and MST_1) is huge.

8.11.1. $\mathbb{Z}_{2^n}$, $t(\alpha) = (2, 2, \ldots, 2)$

Proposition 8.48. *Let $G = \mathbb{Z}_{2^n}$. There are*

$$n! \cdot 2^{\frac{(n-1)n}{2}}$$

canonical logarithmic signatures of type $(2, 2, \ldots, 2)$ for G.

Proof. By Lemma 8.4 we know that here the powers of 2 in the prime factorization of all elements in a logarithmic signature except 0_G are unique.

For the odd element (i.e. the one that does not contain 2 as factor), there are 2^{n-1} possibilities. For the element containing 2 as factor, but not 4, there are 2^{n-2} possibilities. For the element containing 4 as factor, but not 8, there are 2^{n-3} possibilities. And so on. In total, we get $\prod_{k=0}^{n-1} 2^k = 2^{\sum_{k=0}^{n-1} k} = 2^{\frac{(n-1)n}{2}}$ possibilities.

The blocks are freely permutable. This results in $n!$ possibilities.

Combining this, we obtain $n! \cdot 2^{\frac{(n-1)n}{2}}$ different canonical logarithmic signatures. Clearly, all logarithmic signatures (following the structure of Lemma 8.4) are covered. □

8.11.2. $\mathbb{Z}_2^k$, $t(\alpha) = (2, 2, \ldots, 2)$

Proposition 8.49. *Let $G = \mathbb{Z}_2^k$. There are*

$$\prod_{j=0}^{k-1} (2^k - 2^j)$$

canonical logarithmic signatures of type $(2, 2, \ldots, 2)$ for G.

Proof. Observe that G is a vector space over the field $\mathbb{Z}_2$. Thus it is sufficient to compute the number of ordered bases for this vector space.

We construct a basis. For the first element, we can freely choose any vector except the zero vector, i.e. there are $2^k - 1$ possibilities. The second vector may be every vector

except a multiple of the first one (otherwise they would be linear dependent), so there are $2^k - 2$ possibilities. The third vector must not be a linear combination of the first two (these first two generate 4 vectors), so $2^k - 4$ possibilities, and so on. □

9. Dihedral Group

In this section we regard the case when G is a dihedral group.

Motivation. Dihedral groups in some sense are the most simple non-abelian groups. Our goal in this section is to develop factorization and logarithmic signature generation algorithms for dihedral groups.

In Chapter 8, our most commonly used approach was to locate subgroup/periodic blocks and move into the induced factor groups. With dihedral groups, this approach does not work directly, because although even when we encounter a subgroup block, we usually cannot move into the induced factor group, due to most subgroups not being normal (in contrast to abelian groups, where all subgroups are normal).

For example,

$$\alpha = \begin{pmatrix} \mathrm{id} \\ \sigma^{63}\tau \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ \sigma^{56} \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ \sigma^{23}\tau \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ \sigma^{20} \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ \sigma^{67}\tau \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ \sigma^{124}\tau \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ \sigma^{106}\tau \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ \sigma^{10}\tau \end{pmatrix}$$

is a logarithmic signature for $D_{2\cdot 128}$. Although there are plenty of subgroup blocks in α (namely the blocks containing a reflection), none of them is normal.

Furthermore, statically chosen factor groups do not seem to be promising either. For example, even for the rather large normal subgroup $\langle \sigma^{16} \rangle$, none of the non-id elements in α get reduced to id in the factor group $D_{2\cdot 128}/\langle \sigma^{16} \rangle$.

Most structural properties that we have proven for abelian groups do not directly hold for dihedral groups. For example, by Lemma 8.4 we know that every canonical logarithmic signature $\beta \in \Lambda(\mathbb{Z}_{2^{128}})$ of type $t(\beta) = (2, 2, \ldots, 2)$ must contain the element 64 in some block. However, α neither contains σ^{64} nor $\sigma^{64}\tau$.

So, we need to explore the structure of logarithmic signatures for dihedral groups, and develop new methods or extend existing ones to work with dihedral groups.

Our contributions. We first present several interesting block substitution transformations (both unconditional and conditional ones) in dihedral groups, which are later used both in our generation and factorization algorithms. We especially analyze size-permutable blocks in detail.

Subsequently, we design factorization algorithms (some efficient, some not) for various special cases, including the case when all blocks of $\alpha \in \Lambda(D_{2n})$ except one of size 2 contain rotations only (the run-time of our algorithm depends on the run-time of another factorization algorithm for a $\mathbb{Z}_n$ logarithmic signature), $D_{2\cdot p^n}$ (our run-time depends on the structure of the logarithmic signature), the special case $\alpha \in \Lambda(D_{2\cdot 2^n})$ of type $t(\alpha) = (2, 2, \ldots, 2)$ (the algorithm is always efficient, i.e. α is tame; note that α has minimal

length). We slightly generalize the last case by showing that any logarithmic signature $\alpha \in \Lambda(D_{2\cdot 2^n} \times \mathbb{Z}_2^k)$ of type $t(\alpha) = (2, 2, \ldots, 2)$ is tame.

Furthermore, we design and analyze an algorithm to generate logarithmic signatures for D_{2n} with n odd.

Similar to the previous chapter, we close this chapter by counting logarithmic signatures for some specific dihedral groups and logarithmic signature types.

Encoding. For $G = D_{2n}$ with $n \in \mathbb{N}$, we assume that encoding the rotation component of a group element requires $\lceil \log_2 n \rceil$ bits, and that 1 bit is used for encoding whether the element is a reflection, i.e. the code length (number of bits required to store a group element) is

$$b_G = \lceil \log_2 |G| \rceil = \lceil \log_2 n \rceil + 1.$$

Note that this encoding is as compact as possible.

9.1. Transformations

In this section, we have a closer look at transformations applied to logarithmic signatures of dihedral groups.

9.1.1. Block Substitutions

There are two block substitutions that are of great help to simplify logarithmic signatures of dihedral groups. The first one is:

$$\begin{pmatrix} \mathrm{id} \\ \sigma^r\tau \end{pmatrix}\begin{pmatrix} \mathrm{id} \\ \sigma^s \end{pmatrix} \leftrightarrow \begin{pmatrix} \mathrm{id} \\ \sigma^s \end{pmatrix}\begin{pmatrix} \mathrm{id} \\ \sigma^{r-s}\tau \end{pmatrix}.$$

Both block pairs generate the elements $\{\mathrm{id}, \sigma^r\tau, \sigma^s, \sigma^{r-s}\tau\}$. In a logarithmic signature we can therefore substitute blocks of the left form by the blocks on the right (and the other way around) and the result is still a logarithmic signature. Also, if we can compute a factorization in the logarithmic signature with substituted blocks, we can easily derive a factorization in the original signature: the local product of the selected elements in the two substituted blocks just needs to be expressed as a local product in the original signature.

The second interesting block substitution is:

$$\begin{pmatrix} \mathrm{id} \\ \sigma^r\tau \end{pmatrix}\begin{pmatrix} \mathrm{id} \\ \sigma^s\tau \end{pmatrix} \leftrightarrow \begin{pmatrix} \mathrm{id} \\ \sigma^{r-s} \end{pmatrix}\begin{pmatrix} \mathrm{id} \\ \sigma^s\tau \end{pmatrix}.$$

Here, both two blocks generate the elements $\{\mathrm{id}, \sigma^r\tau, \sigma^s\tau, \sigma^{r-s}\}$.

We will later use these two block substitutions to find factorizations in special logarithmic signatures of the dihedral group.

Note that the second substitution can be generalized for larger blocks. If we have a block of size 2 on the right, we can freely transform reflections into rotations and rotations into reflections on the left. An example:

$$\begin{pmatrix} \mathrm{id} \\ \sigma^r\tau \\ \sigma^s\tau \\ \sigma^t\tau \\ \sigma^u\tau \\ \sigma^v\tau \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ \sigma^w\tau \end{pmatrix} \leftrightarrow \begin{pmatrix} \mathrm{id} \\ \sigma^{r-w} \\ \sigma^{s-w} \\ \sigma^{t-w} \\ \sigma^{u-w} \\ \sigma^{v-w} \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ \sigma^w\tau \end{pmatrix} \leftrightarrow \begin{pmatrix} \mathrm{id} \\ \sigma^{r-w} \\ \sigma^s\tau \\ \sigma^{t-w} \\ \sigma^{u-w} \\ \sigma^v\tau \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ \sigma^w\tau \end{pmatrix} \leftrightarrow \begin{pmatrix} \mathrm{id} \\ \sigma^r\tau \\ \sigma^s\tau \\ \sigma^t\tau \\ \sigma^{u-w} \\ \sigma^{v-w} \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ \sigma^w\tau \end{pmatrix}.$$

9.1.2. Conditional Block Substitutions

In the following we have a look at block substitutions that only work under certain circumstances.

$$\begin{pmatrix} \mathrm{id} \\ r_1\tau \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ r_2 \\ r_3 \end{pmatrix} = \{\mathrm{id}, r_1\tau, r_2, r_3, r_1 - r_2\tau, r_1 - r_3\tau\}.$$

- $\begin{pmatrix} \mathrm{id} \\ r_2 \\ r_3 \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ r_1 - r_3\tau \end{pmatrix} = \{\mathrm{id}, r_1\tau, r_2, r_3, r_1 + r_2 - r_3\tau, r_1 - r_3\tau\}$
  is the same if $r_1 - r_2 = r_1 + r_2 - r_3 \Leftrightarrow r_3 = 2r_2$.
- $\begin{pmatrix} \mathrm{id} \\ r_1\tau \\ r_3 \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ r_1 - r_2\tau \end{pmatrix} = \{\mathrm{id}, r_1\tau, r_2, r_3, r_1 - r_2\tau, r_1 - r_2 + r_3\tau\}$
  is the same if $r_1 - r_3 = r_1 - r_2 + r_3 \Leftrightarrow r_2 = 2r_3$.

$$\begin{pmatrix} \mathrm{id} \\ r_1\tau \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ r_2\tau \\ r_3 \end{pmatrix} = \{\mathrm{id}, r_1\tau, r_2\tau, r_3, r_1 - r_2, r_1 - r_3\tau\}.$$

- $\begin{pmatrix} \mathrm{id} \\ r_2\tau \\ r_3 \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ r_1 - r_3\tau \end{pmatrix} = \{\mathrm{id}, r_1\tau, r_2\tau, r_3, -r_1 + r_2 + r_3, r_1 - r_3\tau\}$
  is the same if $r_1 - r_2 = -r_1 + r_2 + r_3 \Leftrightarrow 2r_1 = 2r_2 + r_3$.

$$\begin{pmatrix} \mathrm{id} \\ r_1 \\ r_2\tau \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ r_3 \\ r_4 \end{pmatrix} = \{\mathrm{id}, r_1, r_2\tau, r_3, r_4, r_1 + r_3, r_1 + r_4, r_2 - r_3\tau, r_2 - r_4\tau\}.$$

- $\begin{pmatrix} \mathrm{id} \\ r_3 \\ r_4 \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ r_1 \\ r_2\tau \end{pmatrix} = \{\mathrm{id}, r_1, r_2\tau, r_3, r_4, r_1 + r_3, r_1 + r_4, r_2 + r_4\tau, r_2 + r_3\tau\}$
  is the same if $r_2 - r_3 = r_2 + r_4 \Leftrightarrow r_3 = -r_4$.

- $\begin{pmatrix}\mathrm{id}\\ r_3\\ r_4\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ r_1\\ r_2-r_4\tau\end{pmatrix} = \{\mathrm{id}, r_1, r_2\tau, r_3, r_4, r_1+r_3, r_1+r_4, r_2+r_3-r_4\tau, r_2+r_4\tau\}$
  is the same if $r_2 - r_3 = r_2 + r_3 - r_4 \Leftrightarrow 2r_3 = r_4$.

In the following, we generalize this. We show a connection to antiperiodicity (Section 2.4.2).

Definition 9.1. Let $G = D_{2n}$. Define

$$\varrho : G \times G \to G : (\sigma^d\tau^c, \sigma^r\tau^b) \mapsto \begin{cases}\sigma^r, & \text{if } b = 0,\\ \sigma^{d-r}, & \text{if } b = 1.\end{cases}$$

Naturally, for an $A \subseteq G$ we write $\varrho(g, A) := \{\varrho(g, a) \mid a \in A\}$.

Theorem 9.2. *Let $G = D_{2n}$ and $m \in \mathbb{N}$ odd. Let $T, A \subseteq G$ two canonical sets with $T = \{\mathrm{id}, d\tau\}$ for some $d \in \mathbb{Z}_n$ (i.e. the second element in T is a reflection), $|A| = m$, $|T \cdot A| = 2m$. Set $R := \varrho(d\tau, A)$.*

Then (T, A) is size-permutable if and only if $\mathfrak{Z}(R) \neq \emptyset$.

Proof. T is a subgroup of G; ϱ performs a selective shift using $d\tau$ onto an element in A if and only if the element in A is a reflection; this results in R containing only rotations, and $T \cdot A = T \cdot R$.

We have $T \cdot R = \{g \cdot h \mid g \in T, h \in R\} = T \cup R \cup \{(d-r)\tau \mid r \in R\} = R \cup \{(d-r)\tau \mid r \in R\}$ (due to T and R being canonical).

Assume that canonical sets T' and R' with $|T'| = 2$, $|R'| = m$ and $R' \cdot T' = T \cdot R$ exist. As T' is canonical, we have $\mathrm{id} \in T'$; and as m is odd, the second element in T' must be a reflection ($T \cdot A$ contains exactly as many rotations as reflections; if the second element in T' would be a rotation, R' would have to contain exactly $\frac{m}{2}$ reflections, which is not possible due to m being odd). The only candidates for the reflection in T' are the elements $(d - r)\tau$ (with $r \in R$), because $\mathrm{id} \in R'$. Thus write $T' = \{\mathrm{id}, (d - z)\tau\}$ for a $z \in R$.

W.l.o.g. we can assume that R' contains only rotations, because if R' would contain reflections, we could perform selective shifts using the element $(d-z)\tau$ onto the reflections in R' to produce a set that contains only rotations and still generates the same elements together with T'. As $T \cdot A$ contains exactly m rotations, we must set $R' := R$.

We get $R' \cdot T' = R' \cup \{(r + d - z)\tau \mid r \in R'\} = R \cup \{(r + d - z)\tau \mid r \in R\}$, and thus

$$\begin{aligned} T \cdot R &= R' \cdot T' \\ \Leftrightarrow R \cup \{(d-r)\tau \mid r \in R\} &= R \cup \{(r+d-z)\tau \mid r \in R\} \\ \Leftrightarrow \{(d-r)\tau \mid r \in R\} &= \{(r+d-z)\tau \mid r \in R\} \\ \Leftrightarrow \{-r \mid r \in R\} &= \{r - z \mid r \in R\} \\ \Leftrightarrow R &= \{z - r \mid r \in R\} \\ \Leftrightarrow z &\in \mathfrak{Z}(R). \end{aligned}$$

□

Theorem 9.2 does not necessarily hold if m is even; we give a counter-example in Remark 9.5.

In the proof of Theorem 9.2, z is not necessarily determined uniquely, if it exists. For example, if $R \trianglelefteq G$, then all $z \in R$ result in $R' \cdot T' = T \cdot R$ (i.e. $\mathfrak{Z}(R) = R$).

Example 9.3. Let $G = D_{2\cdot 15}$, $T = \{\mathrm{id}, 7\tau\}$, $A = \{\mathrm{id}, 3, 7, 11\tau, 13\tau\}$.

In order to test whether (T, A) is size-permutable, we first compute $R := \varrho(7\tau, A) = \{\mathrm{id}, 3, 7, 11, 9\} = \{\mathrm{id}, 3, 7, 9, 11\}$. We now test whether there exists at least one $z \in R$ such that $R = \{z - r \mid r \in R\}$:

- $z = 0$: $0 - 3 \bmod 15 = 12 \notin R \Rightarrow z = 0$ is not a valid choice.
- $z = 3$: $3 - 0 \bmod 15 = 3 \in R$, $3 - 3 \bmod 15 = 0 \in R$, $3 - 7 \bmod 15 = 11 \in R$, $3 - 9 \bmod 15 = 9 \in R$, $3 - 11 \bmod 15 = 7 \in R \Rightarrow z = 3$ is a valid choice and with $(d - z)\tau = (7 - 3)\tau = 4\tau$, we get
$$T \cdot A = \{\mathrm{id}, 7\tau\} \cdot \{\mathrm{id}, 3, 7, 11\tau, 13\tau\} = \{\mathrm{id}, 3, 7, 9, 11\} \cdot \{\mathrm{id}, 4\tau\} = R' \cdot T'.$$
- $z = 7$: $7 - 3 = 4 \notin R \Rightarrow z = 7$ is not a valid choice.
- $z = 9$: $9 - 7 = 2 \notin R \Rightarrow z = 9$ is not a valid choice.
- $z = 11$: $11 - 9 = 2 \notin R \Rightarrow z = 11$ is not a valid choice.

So, $\mathfrak{Z}(R) = \{3\} \neq \emptyset$ and (T, A) is size-permutable.

Example 9.4. Let $G = D_{2\cdot 15}$, $T = \{\mathrm{id}, 5\tau\}$, $A = \{\mathrm{id}, 4, 6\tau\}$.

In order to test whether (T, A) is size-permutable, we first compute $R := \varrho(5\tau, A) = \{\mathrm{id}, 4, 14\}$. We now test whether there exists at least one $z \in R$ such that $R = \{z - r \mid r \in R\}$:

- $z = 0$: $0 - 4 \bmod 15 = 11 \notin R \Rightarrow z = 0$ is not a valid choice.
- $z = 4$: $4 - 14 \bmod 15 = 5 \notin R \Rightarrow z = 4$ is not a valid choice.
- $z = 14$: $14 - 4 \bmod 15 = 10 \notin R \Rightarrow z = 14$ is not a valid choice.

So, there exists no $z \in R$ with the required property (we get $\mathfrak{Z}(R) = \emptyset$), thus (T, A) is not size-permutable, i.e. there exist no canonical blocks $R', T' \subseteq G$ with $|R'| = 3$, $|T'| = 2$ and $T \cdot A = R' \cdot T'$.

Remark 9.5. In Theorem 9.2, m being odd is a necessary hypothesis; the statement does not necessarily hold if m is even. For example, let $G = D_{2\cdot 10}$, $m = 6$, $T = \{\mathrm{id}, 8\tau\}$, $A = \{\mathrm{id}, 5, 7, 2\tau, 5\tau, 7\tau\}$, $T \cdot A = \{\mathrm{id}, 5, 7, 2\tau, 5\tau, 7\tau, 8\tau, 3\tau, 1\tau, 6, 3, 1\} = \{\mathrm{id}, 1, 3, 5, 6, 7, 1\tau, 2\tau, 3\tau, 5\tau, 7\tau, 8\tau\}$, $|T \cdot A| = 12 = 2 \cdot 6$, $R = \{\mathrm{id}, 5, 7, 6, 3, 1\} = \{\mathrm{id}, 1, 3, 5, 6, 7\}$, $\mathfrak{Z}(R) = \emptyset$ (e.g. due to $0 - 1 \bmod 10 = 9 \notin R$, $1 - 7 \bmod 10 = 4 \notin R$, $3 - 1 \bmod 10 = 2 \notin R$, $5 - 1 \bmod 10 = 4 \notin R$, $6 - 7 \bmod 10 = 9 \notin R$, $7 - 3 \bmod 10 = 4 \notin R$), but (T, A) *is* size-permutable, e.g. with $A' = \{\mathrm{id}, 5, 7, 1\tau, 3\tau, 8\tau\}$ and $T' = \{\mathrm{id}, 6\}$ we have $A' \cdot T' = \{\mathrm{id}, 5, 7, 1\tau, 3\tau, 8\tau, 6, 1, 3, 5\tau, 7\tau, 2\tau\} = \{\mathrm{id}, 1, 3, 5, 6, 7, 1\tau, 2\tau, 3\tau, 5\tau, 7\tau, 8\tau\} = T \cdot A$.

9.2. Factorization Algorithm for $\mathscr{C}_\tau(E(\alpha)) = 1$

Proposition 9.6. *Let $G = D_{2n}$ and $\alpha = (A_1, \ldots, A_s) \in \Lambda(G)$ canonical. Assume there exists an $1 \leq i \leq s$ such that $|A_i| = 2$ and all other A_j (with $j \neq i$) contain only rotations.*

If all canonical $\beta \in \Lambda(\mathbb{Z}_n)$ of type $t(\beta) = (|A_1|, \ldots, |A_{i-1}|, |A_{i+1}|, \ldots, |A_s|)$ are tame, then α is tame.

Proof. As all blocks except A_i contain rotations only, A_i must contain id (due to being canonical) and a reflection $\sigma^d\tau$ with $d \in \mathbb{Z}_n$ (otherwise we could not generate any reflections).

For a factorization of a rotation, id must be selected in A_i. For a factorization of a reflection, the reflection must be selected in A_i.

Let $\varphi : G \to \mathbb{Z}_n : \sigma^k\tau^b \mapsto k$ and $\beta := (\varphi(A_1), \ldots, \varphi(A_{i-1}), \varphi(A_{i+1}), \ldots, \varphi(A_s))$. As every factorization of a rotation of G requires id to be selected in A_i (and every factorization of a reflection of G requires the reflection to be selected in A_i), we have $\beta \in \Lambda(\mathbb{Z}_n)$ and $t(\beta) = (|A_1|, \ldots, |A_{i-1}|, |A_{i+1}|, \ldots, |A_s|)$.

Let $g \in G$ be an element that we want to factor with respect to α.

- If g is a rotation, select id in A_i, factor $\varphi(g)$ with respect to β and select the corresponding elements in α.

  If β is tame, then this approach is efficient.

- If g is a reflection, we continue as follows. Let $\beta^\diamond := (B_1, \ldots, B_{i-1}, B_{i+1}, \ldots B_s)$ with $B_j := \varphi(A_j)$ for all $1 \leq j < i$ and $B_j := \varphi(A_j^{-1}) = \{\varphi(a^{-1}) \mid a \in A_j\}$ for all $i < j \leq s$. $\beta^\diamond$ is the result of performing selective shifts using the reflection in A_i onto all elements in A_{i+1} (in the result, id must be selected in A_i for the factorization of g), subsequently removing A_i, performing a normalization (this inverts all rotations, because $\sigma^d\tau \cdot \sigma^r \cdot (\sigma^d\tau)^{-1} = \sigma^d\tau \cdot \sigma^r \cdot \sigma^d\tau = \sigma^{d-r-d} = \sigma^{-r}$), extracting and cutting the $\sigma^d\tau$ from the right side (to make the last block canonical) and finally going from G to $\mathbb{Z}_n$. Clearly, $\beta^\diamond \in \Lambda(\mathbb{Z}_n)$.

  For the factorization of g with respect to α, select the reflection in A_i, factor $\varphi(g \cdot (\sigma^d\tau)^{-1})$ (which is $\varphi(g \cdot \sigma^d\tau)$) with respect to $\beta^\diamond$ and select the corresponding elements in α.

  If $\beta^\diamond$ (which is of the same type as β) is tame, then this approach is efficient.

Thus α is tame. □

9.3. Factorization Algorithm for $D_{2 \cdot p^n}$

In the following we present an algorithm to find element factorizations in logarithmic signatures of dihedral groups of order $2 \cdot p^n$, where p is a prime and n is a natural number. The run-time of the algorithm is highly dependent on the structure of the logarithmic signature. In Section 9.3.3, we present run-time examples (including an example where only $O(n^2)$ group operations are required, and an example where exponential time is

required).

For the special case $D_{2\cdot 2^n}$, we develop an efficient factorization algorithm in Section 9.4.

9.3.1. Algorithm

Let $D_{2\cdot p^n}$ be the dihedral group of order $2 \cdot p^n$, *ls* a canonical logarithmic signature for this group, and $\sigma^k\tau^c$ the element to factorize.

The factorization is computed by the following function, when called with the logarithmic signature *ls* (represented by a two-dimensional jagged array, using one-based indices), p the prime number in our group $D_{2\cdot p^n}$, $nRotToFactor = k$, $nCurPow = -1$, *vSelected* a list containing as many -1 as we have blocks in the logarithmic signature (denoted by s), and $bTailTau = false$.

Function `FacRec`(LogSig *ls*, Int p, Int *nRotToFactor*, Int *nCurPow*, List<Int> *vSelected*, Bool *bTailTau*) : List<Int>

1. If *vSelected* does not contain any -1:
   a) Set $gl \leftarrow ls[1][vSelected[1]] \cdot ls[2][vSelected[2]] \cdots ls[s][vSelected[s]]$.
   b) If *bTailTau*: set $gl \leftarrow gl \cdot \tau$.
   c) If $gl = \sigma^k\tau^c$: `return` *vSelected*,
   else: `return` *null*.
2. If $nRotToFactor = 0$: set $nPowToFactor \leftarrow -1$,
   else: set $nPowToFactor \leftarrow$ power of p in the prime factorization of *nRotToFactor*.
3. If $((nPowToFactor < nCurPow)$ and $(nPowToFactor \neq -1))$: `return` *null*.
4. Set $L \leftarrow$ candidate sets (see below).
5. For each candidate set S in L:
   a) Clone $lsNew \leftarrow ls$, $vSelectedNew \leftarrow vSelected$, $bTailTauNew \leftarrow bTailTau$.
   b) Select all elements in S (modifies *lsNew*, *vSelectedNew* and *bTailTauNew*; see below).
   c) Compute all possible selected element products (see below).
   d) For each possible product $\sigma^r\tau^d$:
      i. Set $nRotToFactorNew \leftarrow k - r \bmod p^n$.
      ii. Set List<Int> *vSub* $\leftarrow$ `FacRec`(*lsNew*, p, *nRotToFactorNew*, $nCurPow + 1$, *vSelectedNew*, *bTailTauNew*).
      iii. If $vSub \neq null$: `return` *vSub*.

6. `Return` *null*.

Details:

- **Finding candidate sets L.** We call a list of element positions (block and index) a *candidate set* S, when the following conditions are fulfilled:
  - For each block there is at most one position specifying an index in this block (i.e. block indices are unique in S).
  - For each position (x, y) the power of p in the power of σ of the element in block x at index y in the logarithmic signature is exactly *nCurPow*.
  - The number of elements in the candidate set is even when $nPowToFactor \neq nCurPow$, or odd when $nPowToFactor = nCurPow$.

  L is the set of all possible candidate sets S.

  Example. Let *ls* be the following logarithmic signature of $D_{2\cdot 2^8}$:

  $$\binom{\mathrm{id}}{109\tau}\binom{\mathrm{id}}{2^4\cdot 3\cdot 5}\binom{\mathrm{id}}{2^2\cdot 3\cdot 11\tau}\binom{\mathrm{id}}{2^2\cdot 3\cdot 13}\binom{\mathrm{id}}{2^6}\binom{\mathrm{id}}{2^5}\binom{\mathrm{id}}{2\cdot 3\cdot 11\tau}\binom{\mathrm{id}}{2\cdot 97\tau}\binom{\mathrm{id}}{2^3\cdot 3^3}.$$

  Here, p is 2. Let *nPowToFactor* be 3 and *nCurPow* be 2 (i.e. we are looking at elements with rotations containing 2 to the power of 2). As we have $nPowToFactor \neq nCurPow$, the candidate sets must contain an even number of elements. So, $L = \{\emptyset, \{(3, 2), (4, 2)\}\}$. This means that in order to obtain an even number of elements containing 2^2 (exactly) as factor in rotations, we can either select no elements at all or select both $2^2 \cdot 3 \cdot 11\tau$ and $2^2 \cdot 3 \cdot 13$.

- **Selecting elements of a candidate set S.** In order to select elements of S, we do the following for each element position $(x, y) \in S$ (ordered ascendingly by x):
  1. Select the element in *vSelectedNew*, i.e. set $vSelectedNew[x] \leftarrow y$.
  2. If the element at (x, y) in *ls* is a reflection:
     a) Multiply the element at (x, y) in *ls* with τ from the right.
     b) For each block $ls[i]$ with $i > x$ (ascendingly):
        - For each element $g_j = \sigma^{r_j}\tau^{c_j}$ in $ls[i]$:
          * Replace g_j by $\sigma^{-r_j}\tau^{c_j}$ (in $ls[i]$).
     c) Toggle *bTailTauNew*.

  Example. Let *ls* be the following logarithmic signature of $D_{2\cdot 2^8}$:

  $$\binom{\mathrm{id}}{109\tau}\binom{\mathrm{id}}{2^4\cdot 3\cdot 5}\binom{\mathrm{id}}{2^2\cdot 3\cdot 11\tau}\binom{\mathrm{id}}{2^2\cdot 3\cdot 13}\binom{\mathrm{id}}{2^6}\binom{\mathrm{id}}{2^5}\binom{\mathrm{id}}{2\cdot 3\cdot 11\tau}\binom{\mathrm{id}}{2\cdot 97\tau}\binom{\mathrm{id}}{2^3\cdot 3^3}.$$

From now on, we mark selected elements by rectangles. If *bTailTauNew* is *true*, we indicate this by appending a block containing only the element τ.

By selecting the element $2 \cdot 3 \cdot 11\tau$, we would obtain the following logarithmic signature:

$$\left(\begin{smallmatrix}\mathrm{id}\\ 109\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\boxed{\mathrm{id}}\\ \boxed{2\cdot 3\cdot 11}\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 31\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 5\end{smallmatrix}\right)(\tau).$$

- **Computing possible selected element products.** Having selected some elements of the logarithmic signature, we compute the product of these selected elements. However, depending on whether we might later select a reflection between these elements (i.e. there is a block without any selected element and at least one element is a reflection), the rotations of the following selected elements either have to be added or subtracted, i.e. there are two cases / possible products. In this step of the algorithm, we compute all of these possible products.

  Example. Let *ls* be the following logarithmic signature of $D_{2\cdot 2^8}$ (selected elements are marked using rectangles):

$$\left(\begin{smallmatrix}\mathrm{id}\\ 109\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ \boxed{2^4\cdot 3\cdot 5}\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ \boxed{2^2\cdot 3\cdot 11\tau}\end{smallmatrix}\right)\left(\begin{smallmatrix}\boxed{\mathrm{id}}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\boxed{\mathrm{id}}\\ 2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 97\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ \boxed{2^3\cdot 3^3}\end{smallmatrix}\right).$$

  Here, there are four possible products. The first block does not have any selected element and contains a reflection, giving us two cases. The fifth and sixth blocks do not have selected elements, but also do not contain any reflections, so we can ignore these blocks. The eighth block does not have any selection and contains a reflection, giving us two more cases. So, we have the following four possible products:

  Case $(\mathrm{id}, \mathrm{id})$**:** $\sigma^{2^4\cdot 3\cdot 5}\cdot\sigma^{2^2\cdot 3\cdot 11}\tau\cdot\mathrm{id}\cdot\mathrm{id}\cdot\sigma^{2^3\cdot 3^3} = \sigma^{2^4\cdot 3\cdot 5+2^2\cdot 3\cdot 11-2^3\cdot 3^3}\tau = \sigma^{2^2\cdot 3\cdot 13}\tau.$

  Case (τ, id)**:** $\sigma^{-2^4\cdot 3\cdot 5}\cdot\sigma^{-2^2\cdot 3\cdot 11}\tau\cdot\mathrm{id}\cdot\mathrm{id}\cdot\sigma^{-2^3\cdot 3^3} = \sigma^{-2^4\cdot 3\cdot 5-2^2\cdot 3\cdot 11+2^3\cdot 3^3}\tau = \sigma^{2^2\cdot 5^2}\tau.$

  Case (id, τ)**:** $\sigma^{2^4\cdot 3\cdot 5}\cdot\sigma^{2^2\cdot 3\cdot 11}\tau\cdot\mathrm{id}\cdot\mathrm{id}\cdot\sigma^{-2^3\cdot 3^3} = \sigma^{2^4\cdot 3\cdot 5+2^2\cdot 3\cdot 11+2^3\cdot 3^3}\tau = \sigma^{2^2\cdot 19}\tau.$

  Case (τ, τ)**:** $\sigma^{-2^4\cdot 3\cdot 5}\cdot\sigma^{-2^2\cdot 3\cdot 11}\tau\cdot\mathrm{id}\cdot\mathrm{id}\cdot\sigma^{2^3\cdot 3^3} = \sigma^{-2^4\cdot 3\cdot 5-2^2\cdot 3\cdot 11-2^3\cdot 3^3}\tau = \sigma^{2^2\cdot 3^2\cdot 5}\tau.$

9.3.2. Example

In this example we show how the algorithm in Section 9.3.1 finds an element factorization in a given canonical logarithmic signature.

Let *ls* be the following logarithmic signature of $D_{2\cdot 2^8}$:

$$\left(\begin{smallmatrix}\mathrm{id}\\ 109\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 97\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3^3\end{smallmatrix}\right).$$

We are looking for the factorization of σ^{42}.

On the topmost recursion level of `FacRec`, only τ can be selected (because the rotation component has to be zero and the element must not be the identity). However, as there is no single τ (without a rotation) in the logarithmic signature, the function calls itself recursively, now with *nCurPow* + 1, i.e. 0.

The only element (except id) containing $p = 2$ to the power of 0 (i.e. 2 does not appear in the prime factorization of the rotation component of the dihedral group element) is $\sigma^{109}\tau$. As we want to factorize σ^{42}, an even number, σ^{109} cannot be part of the selection. So, id is selected in this block, and the function calls itself with *nCurPow* = 1 and the following signature:

$$(\boxed{\tfrac{\mathrm{id}}{109\tau}})(\tfrac{\mathrm{id}}{2^4\cdot 3\cdot 5})(\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 11\tau})(\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 13})(\tfrac{\mathrm{id}}{2^6})(\tfrac{\mathrm{id}}{2^5})(\tfrac{\mathrm{id}}{2\cdot 3\cdot 11\tau})(\tfrac{\mathrm{id}}{2\cdot 97\tau})(\tfrac{\mathrm{id}}{2^3\cdot 3^3}).$$

Now we are looking for elements with $p = 2$ to the power of 1. These are $\sigma^{2\cdot 3\cdot 11}\tau$ and $\sigma^{2\cdot 97}\tau$. As $42 = 2\cdot 3\cdot 7$ contains 2 to the power of 1, we need to select an odd number of these elements. Possible selections therefore are (with the subsequent element transformation algorithm already applied):

$$(\boxed{\tfrac{\mathrm{id}}{109\tau}})(\tfrac{\mathrm{id}}{2^4\cdot 3\cdot 5})(\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 11\tau})(\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 13})(\tfrac{\mathrm{id}}{2^6})(\tfrac{\mathrm{id}}{2^5})(\tfrac{\mathrm{id}}{\boxed{2\cdot 3\cdot 11}})(\boxed{\tfrac{\mathrm{id}}{2\cdot 31\tau}})(\tfrac{\mathrm{id}}{2^3\cdot 5})(\tau),$$
$$(\boxed{\tfrac{\mathrm{id}}{109\tau}})(\tfrac{\mathrm{id}}{2^4\cdot 3\cdot 5})(\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 11\tau})(\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 13})(\tfrac{\mathrm{id}}{2^6})(\tfrac{\mathrm{id}}{2^5})(\boxed{\tfrac{\mathrm{id}}{2\cdot 3\cdot 11\tau}})(\tfrac{\mathrm{id}}{\boxed{2\cdot 97}})(\tfrac{\mathrm{id}}{2^3\cdot 5})(\tau).$$

The function would call itself recursively for both of these possibilities. In order to keep this example short, we continue only looking at the possibility leading to the solution (the other possibility results in no solution, as expected), which is the second one. So, we select id in block 7 and $\sigma^{2\cdot 97}\tau$ in block 8.

The next step is to compute the remaining number to factorize. For this, we need to compute the possible selected element products. We have two possible products, because there is at least one reflection in an unselected block in front of block 8; these are: $\sigma^{2\cdot 97}\tau$ and $\sigma^{-2\cdot 97}\tau$. Like above, the algorithm would try both of these possibilities, but to shorten the example, we magically know that the second possibility will lead to the solution.

We calculate the remaining rotation to factorize: $-2\cdot 97 + x = 42 \Leftrightarrow x = 42 + 2\cdot 97 = 236 = 2^2\cdot 59$.

Now $p = 2$ to the power of 2. As $2^2\cdot 59$ contains 2 to the power of 2, we need to select an odd number of elements with rotations containing 2^2. These are $\sigma^{2^2\cdot 3\cdot 11}\tau$ and $\sigma^{2^2\cdot 3\cdot 13}$. This gives us two possibilities:

$$(\boxed{\tfrac{\mathrm{id}}{109\tau}})(\tfrac{\mathrm{id}}{2^4\cdot 3\cdot 5})(\tfrac{\mathrm{id}}{\boxed{2^2\cdot 3\cdot 11}})(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 5^2}})(\tfrac{\mathrm{id}}{2^6\cdot 3})(\tfrac{\mathrm{id}}{2^5\cdot 7})(\boxed{\tfrac{\mathrm{id}}{2\cdot 5\cdot 19\tau}})(\tfrac{\mathrm{id}}{\boxed{2\cdot 31}})(\tfrac{\mathrm{id}}{2^3\cdot 3^3}),$$
$$(\boxed{\tfrac{\mathrm{id}}{109\tau}})(\tfrac{\mathrm{id}}{2^4\cdot 3\cdot 5})(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 11\tau}})(\tfrac{\mathrm{id}}{\boxed{2^2\cdot 3\cdot 13}})(\tfrac{\mathrm{id}}{2^6})(\tfrac{\mathrm{id}}{2^5})(\boxed{\tfrac{\mathrm{id}}{2\cdot 3\cdot 11\tau}})(\tfrac{\mathrm{id}}{\boxed{2\cdot 97}})(\tfrac{\mathrm{id}}{2^3\cdot 5})(\tau).$$

The algorithm would try both; we try the first one first. As there are no free reflections anymore, there is only exactly one possible product: $\sigma^{2^2\cdot 3\cdot 11}\sigma^{2\cdot 31} = \sigma^{2^2\cdot 3\cdot 11+2\cdot 31} = \sigma^{194} = \sigma^{2\cdot 97}$. So, the rotation we still need to factorize is $2\cdot 97 + x = 42 \Leftrightarrow x = 42 - 2\cdot 97 = -152 \equiv 104 = 2^3\cdot 13 \pmod{2^8}$.

The only free element with a rotation containing 2 to the power of 3 is $2^3\cdot 3^3$, so this gets selected:

$$\left(\boxed{\tfrac{\mathrm{id}}{109\tau}}\right)\left(\tfrac{\mathrm{id}}{2^4\cdot 3\cdot 5}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 11}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 5^2}}\right)\left(\tfrac{\mathrm{id}}{2^6\cdot 3}\right)\left(\tfrac{\mathrm{id}}{2^5\cdot 7}\right)\left(\boxed{\tfrac{\mathrm{id}}{2\cdot 5\cdot 19\tau}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2\cdot 31}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^3\cdot 3^3}}\right).$$

The unique product is $\sigma^{2^2\cdot 3\cdot 11}\sigma^{2\cdot 31}\sigma^{2^3\cdot 3^3} = \sigma^{2^2\cdot 3\cdot 11+2\cdot 31+2^3\cdot 3^3} = \sigma^{410} = \sigma^{154} = \sigma^{2\cdot 7\cdot 11}$. Thus we still need to factorize $2\cdot 7\cdot 11 + x = 42 \Leftrightarrow x = 42 - 2\cdot 7\cdot 11 = -112 \equiv 144 = 2^4\cdot 3^2 \pmod{2^8}$. There is only one possibility:

$$\left(\boxed{\tfrac{\mathrm{id}}{109\tau}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^4\cdot 3\cdot 5}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 11}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 5^2}}\right)\left(\tfrac{\mathrm{id}}{2^6\cdot 3}\right)\left(\tfrac{\mathrm{id}}{2^5\cdot 7}\right)\left(\boxed{\tfrac{\mathrm{id}}{2\cdot 5\cdot 19\tau}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2\cdot 31}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^3\cdot 3^3}}\right).$$

This time the unique product is $\sigma^{2^4\cdot 3\cdot 5}\sigma^{2^2\cdot 3\cdot 11}\sigma^{2\cdot 31}\sigma^{2^3\cdot 3^3} = \sigma^{650} = \sigma^{138} = \sigma^{2\cdot 3\cdot 23}$. Now we need to factorize $2\cdot 3\cdot 23 + x = 42 \Leftrightarrow x = 42 - 2\cdot 3\cdot 23 = -96 \equiv 160 = 2^5\cdot 5 \pmod{2^8}$. Again, there is only one possibility:

$$\left(\boxed{\tfrac{\mathrm{id}}{109\tau}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^4\cdot 3\cdot 5}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 11}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 5^2}}\right)\left(\tfrac{\mathrm{id}}{2^6\cdot 3}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^5\cdot 7}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2\cdot 5\cdot 19\tau}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2\cdot 31}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^3\cdot 3^3}}\right).$$

This results in a unique product $\sigma^{2^4\cdot 3\cdot 5}\sigma^{2^2\cdot 3\cdot 11}\sigma^{2^5\cdot 7}\sigma^{2\cdot 31}\sigma^{2^3\cdot 3^3} = \sigma^{874} = \sigma^{106} = 2\cdot 53$. So, now we are looking for $2\cdot 53 + x = 42 \Leftrightarrow x = 42 - 2\cdot 53 = -64 \equiv 192 = 2^6\cdot 3 \pmod{2^8}$. As before, there is only one possibility:

$$\left(\boxed{\tfrac{\mathrm{id}}{109\tau}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^4\cdot 3\cdot 5}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 11}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 5^2}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^6\cdot 3}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^5\cdot 7}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2\cdot 5\cdot 19\tau}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2\cdot 31}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^3\cdot 3^3}}\right).$$

Let us compute the (unique) product of the selected elements:

$$\mathrm{id}\cdot\sigma^{2^4\cdot 3\cdot 5}\cdot\sigma^{2^2\cdot 3\cdot 11}\cdot\mathrm{id}\cdot\sigma^{2^6\cdot 3}\cdot\sigma^{2^5\cdot 7}\cdot\mathrm{id}\cdot\sigma^{2\cdot 31}\cdot\sigma^{2^3\cdot 3^3} = \sigma^{1066} = \sigma^{42}.$$

This is the element that we were looking for! In each block one element has been selected, thus the algorithm terminates (passing the factorization up through *vSub*).

Note that the logarithmic signature has changed in the process (the logarithmic signature above is not the same as the original one), but the element positions still lead to a factorization in the original logarithmic signature. This is because the transformations did not change the factorization for this product. We can verify this in our example by computing the product of the elements in the original logarithmic signature:

$$\left(\boxed{\tfrac{\mathrm{id}}{109\tau}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^4\cdot 3\cdot 5}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 11\tau}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^2\cdot 3\cdot 13}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^6}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^5}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2\cdot 3\cdot 11\tau}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2\cdot 97\tau}}\right)\left(\boxed{\tfrac{\mathrm{id}}{2^3\cdot 3^3}}\right).$$

The product of the selected elements is

$$\begin{aligned}
&\mathrm{id}\cdot\sigma^{2^4\cdot 3\cdot 5}\cdot\sigma^{2^2\cdot 3\cdot 11}\tau\cdot\mathrm{id}\cdot\sigma^{2^6}\cdot\sigma^{2^5}\cdot\mathrm{id}\cdot\sigma^{2\cdot 97}\tau\cdot\sigma^{2^3\cdot 3^3}\\
&=\sigma^{2^4\cdot 3\cdot 5}\cdot\sigma^{2^2\cdot 3\cdot 11}\cdot\sigma^{-2^6}\cdot\sigma^{-2^5}\cdot\sigma^{-2\cdot 97}\cdot\sigma^{2^3\cdot 3^3}\\
&=\sigma^{298}\\
&=\sigma^{42}.
\end{aligned}$$

9.3.3. Run-Time Examples

Example for $O(n^2)$ group operations. We present a family of logarithmic signatures for $D_{2\cdot 2^n}$, in which the algorithm finds factorizations using $O(n^2)$ group operations.

Let $n\in\mathbb{N}$ and ls the following logarithmic signature for $D_{2\cdot 2^n}$:

$$\tbinom{\mathrm{id}}{2^0}\tbinom{\mathrm{id}}{2^1}\tbinom{\mathrm{id}}{2^2}\tbinom{\mathrm{id}}{2^3}\cdots\tbinom{\mathrm{id}}{2^{n-2}}\tbinom{\mathrm{id}}{2^{n-1}}\tbinom{\mathrm{id}}{\tau}.$$

It is clear that this is a logarithmic signature for $D_{2\cdot 2^n}$: using the first $s-1$ blocks (the signature has $s=n+1$ blocks) we can generate all rotations (each number $0\le i<2^n$ can be written uniquely as $\sum_{j=0}^{n-1}x_j2^j$ with $x_j\in\{0,1\}$ for all j), and the last block on the one hand makes the signature generate all rotations (using id) and on the other hand all reflections (using τ).

The algorithm requires n recursive calls for such a signature. In each call there is exactly one candidate set in L, because there is only one element in the logarithmic signature whose rotation contains $p=2$ to the power of *nCurPow*. To recognize this, the algorithm requires $O(n)$ group operations, because all blocks are investigated. The procedure to select an element requires $O(1)$ group operations, because there are no blocks right of the block containing τ and none of the other blocks contain reflections. For each candidate set there is exactly one possible selected element product, because the first n blocks contain only rotations; computing this product requires $O(n)$ group operations. All in all, the algorithm requires $O(n)\cdot(O(n)+O(n))=O(n^2)$ group operations to find the factorization of an element.

Example for exponential run-time. Now we present a family of logarithmic signatures for $D_{2\cdot 2^n}$ with n even, in which the algorithm can require exponential time to find factorizations of elements.

Let $n\in\mathbb{N}$. We start with a signature that obviously is a logarithmic signature for $D_{2\cdot 2^n}$ and then successively apply the second block substitution in Section 9.1.1 from right to left:

$$\begin{aligned}
&\tbinom{\mathrm{id}}{2^{n-1}}\tbinom{\mathrm{id}}{2^{n-2}}\tbinom{\mathrm{id}}{2^{n-3}}\cdots\tbinom{\mathrm{id}}{2^2}\tbinom{\mathrm{id}}{2^1}\tbinom{\mathrm{id}}{2^0}\tbinom{\mathrm{id}}{\tau}\\
\leftrightarrow\ &\tbinom{\mathrm{id}}{2^{n-1}}\tbinom{\mathrm{id}}{2^{n-2}}\tbinom{\mathrm{id}}{2^{n-3}}\cdots\tbinom{\mathrm{id}}{2^1}\tbinom{\mathrm{id}}{(2^0+0)\tau}\tbinom{\mathrm{id}}{\tau}\\
\leftrightarrow\ &\tbinom{\mathrm{id}}{2^{n-1}}\tbinom{\mathrm{id}}{2^{n-2}}\tbinom{\mathrm{id}}{2^{n-3}}\cdots\tbinom{\mathrm{id}}{2^2}\tbinom{\mathrm{id}}{(2^1+2^0)\tau}\tbinom{\mathrm{id}}{2^0\tau}\tbinom{\mathrm{id}}{\tau}
\end{aligned}$$

$$
\begin{aligned}
&\leftrightarrow \left(\begin{smallmatrix}\mathrm{id}\\ 2^{n-1}\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^{n-2}\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^{n-3}\end{smallmatrix}\right)\cdots\left(\begin{smallmatrix}\mathrm{id}\\ 2^{3}\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (2^2+2^1+2^0)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (2^1+2^0)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^0\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ \tau\end{smallmatrix}\right)\\
&\qquad\qquad\vdots\\
&\leftrightarrow \left(\begin{smallmatrix}\mathrm{id}\\ 2^{n-1}\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^{n-2}\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (\sum_{i=0}^{n-3}2^i)\tau\end{smallmatrix}\right)\cdots\left(\begin{smallmatrix}\mathrm{id}\\ (2^2+2^1+2^0)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (2^1+2^0)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^0\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ \tau\end{smallmatrix}\right)\\
&\leftrightarrow \left(\begin{smallmatrix}\mathrm{id}\\ 2^{n-1}\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (\sum_{i=0}^{n-2}2^i)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (\sum_{i=0}^{n-3}2^i)\tau\end{smallmatrix}\right)\cdots\left(\begin{smallmatrix}\mathrm{id}\\ (2^2+2^1+2^0)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (2^1+2^0)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^0\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ \tau\end{smallmatrix}\right)\\
&\leftrightarrow \left(\begin{smallmatrix}\mathrm{id}\\ (\sum_{i=0}^{n-1}2^i)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (\sum_{i=0}^{n-2}2^i)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (\sum_{i=0}^{n-3}2^i)\tau\end{smallmatrix}\right)\cdots\left(\begin{smallmatrix}\mathrm{id}\\ (2^2+2^1+2^0)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (2^1+2^0)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^0\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ \tau\end{smallmatrix}\right)\\
&= \left(\begin{smallmatrix}\mathrm{id}\\ (2^n-1)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (2^{n-1}-1)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (2^{n-2}-1)\tau\end{smallmatrix}\right)\cdots\left(\begin{smallmatrix}\mathrm{id}\\ (2^3-1)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (2^2-1)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ (2^1-1)\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ \tau\end{smallmatrix}\right).
\end{aligned}
$$

This is a logarithmic signature for $D_{2\cdot 2^n}$, because the one that we started with obviously was one and each two substituted blocks locally generate the same four elements as the original two blocks.

In the first recursive call, i.e. when looking at $p = 2$ to the power of 0, L contains $\frac{2^n}{2} = 2^{n-1}$ candidate sets, because all of the first n (free) blocks contain an element with an odd rotation component (i.e. with $p = 2$ to the power of 0). As the loop iterates over all of these candidate sets and only one leads to the solution, the run-time all in all is exponential.

9.4. Factoring in $D_{2\cdot 2^n}$

In the following we develop an algorithm to efficiently find element factorizations with respect to logarithmic signatures of type $(2, 2, \ldots, 2)$ in $D_{2\cdot 2^n}$.

The basic idea is to first simplify the given logarithmic signature (using the τ-reduction transformation in Section 9.4.1), compute a factorization in the simplified signature (which will be easy), and finally derive the factorization in the original signature by undoing the τ-reduction transformation.

9.4.1. τ-Reduction Transformation

The goal of this transformation is to minimize the number of reflection elements in the logarithmic signature. In type $(2, 2, \ldots, 2)$ signatures of $D_{2\cdot 2^n}$ we can use the two block substitutions from Section 9.1.1 to reduce the number of reflections to exactly one:

Function `TauReduce`(LogSig *ls*, List<UInt> *vSubstApplied*) : Void

- For $i \leftarrow 1$ to $|ls| - 1$:
  - If $ls[i][2]$ is a reflection:
    - If $ls[i+1][2]$ is a reflection:
      1. Set $ls[i][2] \leftarrow ls[i][2] \cdot ls[i+1][2]$.
      2. *vSubstApplied*.`Append`(2).

      else:

1. Set $gl \leftarrow ls[i+1][2]$.
2. Set $gr \leftarrow ls[i][2] \cdot ls[i+1][2]$.
3. Set $ls[i][2] \leftarrow gl$.
4. Set $ls[i+1][2] \leftarrow gr$.
5. *vSubstApplied*.`Append`(1).

else: *vSubstApplied*.`Append`(0).

This function transforms the given logarithmic signature in-place and additionally saves for each block whether a substitution has been applied or not into the *vSubstApplied* list (we need this later when undoing the transformation and reconstructing the original signature).

9.4.2. Algorithm

Algorithm 9.7. Let ls be a type $(2, 2, \ldots, 2)$ canonical logarithmic signature of $D_{2\cdot 2^n}$. The following algorithm computes the factorization (block indices) for an element g:

Function `FactorizeD2n`(LogSig ls, GroupElement g) : List<UInt>

1. Let $n \leftarrow |ls|$.
2. Let List<UInt> *vFactorization* ← `new` List<UInt>(n, 1).
3. Let List<UInt> *vSubstApplied* ← `new` List<UInt>().
4. `TauReduce`(ls, *vSubstApplied*).
5. If g is a reflection:
   a) Set $vFactorization[n] \leftarrow 2$.
   b) Set $g \leftarrow g \cdot ls[n][2]^{-1}$.
6. Let Int $nPow \leftarrow 0$.
7. While $g \neq$ id:
   a) Find the element $h \neq$ id in the first $n-1$ blocks of ls in whose prime factorization 2 appears to the power of $nPow$ (this can be done in constant time, see Section 9.4.4).

      Let x be the index of the block containing h.
   b) If 2 appears to the power of $nPow$ in g:
      i. Set $vFactorization[x] \leftarrow 2$.
      ii. Set $g \leftarrow g \cdot h^{-1}$.
   c) Set $nPow \leftarrow nPow + 1$.

8. For Int $t \leftarrow n-1$ down to 1:
   a) If $vSubstApplied[t] = 1$:
      i. Set $gl \leftarrow ls[t][2]$.
      ii. Set $gr \leftarrow ls[t+1][2]$.
      iii. Set $ls[t][2] \leftarrow gl \cdot gr$.
      iv. Set $ls[t+1][2] \leftarrow gl$.
      v. If $vFactorization[t] = 2 \wedge vFactorization[t+1] = 1$:
         set $vFactorization[t] \leftarrow 1$ and $vFactorization[t+1] \leftarrow 2$,
      else if $vFactorization[t] = 1 \wedge vFactorization[t+1] = 2$:
         set $vFactorization[t] \leftarrow 2$ and $vFactorization[t+1] \leftarrow 2$,
      else if $vFactorization[t] = 2 \wedge vFactorization[t+1] = 2$:
         set $vFactorization[t] \leftarrow 2$ and $vFactorization[t+1] \leftarrow 1$.
   else if $vSubstApplied[t] = 2$:
      i. Set $ls[t][2] \leftarrow ls[t][2] \cdot ls[t+1][2]$.
      ii. If $vFactorization[t] = 2 \wedge vFactorization[t+1] = 1$:
         set $vFactorization[t] \leftarrow 2$ and $vFactorization[t+1] \leftarrow 2$,
      else if $vFactorization[t] = 2 \wedge vFactorization[t+1] = 2$:
         set $vFactorization[t] \leftarrow 2$ and $vFactorization[t+1] \leftarrow 1$.
9. `Return` *vFactorization*.

9.4.3. Example

Let ls be the following logarithmic signature of $D_{2\cdot 2^8}$ (the same as in Section 9.3.2):

$$\left(\begin{smallmatrix}\mathrm{id}\\ 109\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 97\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3^3\end{smallmatrix}\right).$$

We are looking for the factorization of σ^{42}.

First of all, we compute the simplified logarithmic signature (`TauReduce`):

i	ls	$vSubstApplied$
	$\left(\begin{smallmatrix}\mathrm{id}\\ 109\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 97\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3^3\end{smallmatrix}\right)$	
1	$\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 5^3\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 97\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3^3\end{smallmatrix}\right)$	1
2	$\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 3\cdot 83\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 97\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3^3\end{smallmatrix}\right)$	2
3	$\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 3\cdot 83\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 29\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 97\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3^3\end{smallmatrix}\right)$	1
4	$\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 3\cdot 83\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3\cdot 7\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 97\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3^3\end{smallmatrix}\right)$	1
5	$\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 3\cdot 83\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 17\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 97\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3^3\end{smallmatrix}\right)$	1
6	$\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 3\cdot 83\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 5\cdot 7\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 3\cdot 11\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 97\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3^3\end{smallmatrix}\right)$	2
7	$\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 3\cdot 83\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 5\cdot 7\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^7\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 97\tau\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3^3\end{smallmatrix}\right)$	2
8	$\left(\begin{smallmatrix}\mathrm{id}\\ 2^4\cdot 3\cdot 5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 3\cdot 83\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^2\cdot 3\cdot 13\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^6\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^5\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 5\cdot 7\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^7\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2^3\cdot 3^3\end{smallmatrix}\right)\left(\begin{smallmatrix}\mathrm{id}\\ 2\cdot 3^2\cdot 13\tau\end{smallmatrix}\right)$	1

$g = \sigma^{42}$ is not a reflection, so the next conditional code section is not executed. Now we compute the factorization in the simplified logarithmic signature (the "While" loop):

nPow	g	h	x	*vFactorization*$[x]$	New g
0	$\sigma^{42} = \sigma^{2\cdot 3\cdot 7}$	$\sigma^{3\cdot 83}$	2	1	σ^{42}
1	$\sigma^{42} = \sigma^{2\cdot 3\cdot 7}$	$\sigma^{2\cdot 5\cdot 7}$	6	2	σ^{228}
2	$\sigma^{228} = \sigma^{2^2\cdot 3\cdot 19}$	$\sigma^{2^2\cdot 3\cdot 13}$	3	2	σ^{72}
3	$\sigma^{72} = \sigma^{2^3\cdot 3^2}$	$\sigma^{2^3\cdot 3^3}$	8	2	σ^{112}
4	$\sigma^{112} = \sigma^{2^4\cdot 7}$	$\sigma^{2^4\cdot 3\cdot 5}$	1	2	σ^{128}
5	$\sigma^{128} = \sigma^{2^7}$	σ^{2^5}	5	1	σ^{128}
6	$\sigma^{128} = \sigma^{2^7}$	σ^{2^6}	4	1	σ^{128}
7	$\sigma^{128} = \sigma^{2^7}$	σ^{2^7}	7	2	id

Visualizing *vFactorization*, we have selected the following elements in the simplified logarithmic signature:

$$\left(\boxed{\substack{\mathrm{id}\\ 2^4\cdot 3\cdot 5}}\right)\left(\boxed{\substack{\mathrm{id}\\ 3\cdot 83}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 13}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2^6}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2^5}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2\cdot 5\cdot 7}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2^7}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2^3\cdot 3^3}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2\cdot 3^2\cdot 13\tau}}\right).$$

Computing the product of the selected elements indeed results in σ^{42}. The rest of the algorithm undoes the τ-reduction transformation in order to find a factorization in the original logarithmic signature. Recall that $vSubstApplied = (1, 2, 1, 1, 1, 2, 2, 1)$.

t	New *ls*	New *vFactorization*
	$\left(\substack{\mathrm{id}\\ 2^4\cdot 3\cdot 5}\right)\left(\substack{\mathrm{id}\\ 3\cdot 83}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 13}\right)\left(\substack{\mathrm{id}\\ 2^6}\right)\left(\substack{\mathrm{id}\\ 2^5}\right)\left(\substack{\mathrm{id}\\ 2\cdot 5\cdot 7}\right)\left(\substack{\mathrm{id}\\ 2^7}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 3^3}\right)\left(\substack{\mathrm{id}\\ 2\cdot 3^2\cdot 13\tau}\right)$	$(2, 1, 2, 1, 1, 2, 2, 2, 1)$
8	$\left(\substack{\mathrm{id}\\ 2^4\cdot 3\cdot 5}\right)\left(\substack{\mathrm{id}\\ 3\cdot 83}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 13}\right)\left(\substack{\mathrm{id}\\ 2^6}\right)\left(\substack{\mathrm{id}\\ 2^5}\right)\left(\substack{\mathrm{id}\\ 2\cdot 5\cdot 7}\right)\left(\substack{\mathrm{id}\\ 2^7}\right)\left(\substack{\mathrm{id}\\ 2\cdot 97\tau}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 3^3}\right)$	$(2, 1, 2, 1, 1, 2, 2, 1, 2)$
7	$\left(\substack{\mathrm{id}\\ 2^4\cdot 3\cdot 5}\right)\left(\substack{\mathrm{id}\\ 3\cdot 83}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 13}\right)\left(\substack{\mathrm{id}\\ 2^6}\right)\left(\substack{\mathrm{id}\\ 2^5}\right)\left(\substack{\mathrm{id}\\ 2\cdot 5\cdot 7}\right)\left(\substack{\mathrm{id}\\ 2\cdot 3\cdot 11\tau}\right)\left(\substack{\mathrm{id}\\ 2\cdot 97\tau}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 3^3}\right)$	$(2, 1, 2, 1, 1, 2, 2, 2, 2)$
6	$\left(\substack{\mathrm{id}\\ 2^4\cdot 3\cdot 5}\right)\left(\substack{\mathrm{id}\\ 3\cdot 83}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 13}\right)\left(\substack{\mathrm{id}\\ 2^6}\right)\left(\substack{\mathrm{id}\\ 2^5}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 17\tau}\right)\left(\substack{\mathrm{id}\\ 2\cdot 3\cdot 11\tau}\right)\left(\substack{\mathrm{id}\\ 2\cdot 97\tau}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 3^3}\right)$	$(2, 1, 2, 1, 1, 2, 1, 2, 2)$
5	$\left(\substack{\mathrm{id}\\ 2^4\cdot 3\cdot 5}\right)\left(\substack{\mathrm{id}\\ 3\cdot 83}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 13}\right)\left(\substack{\mathrm{id}\\ 2^6}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 3\cdot 7\tau}\right)\left(\substack{\mathrm{id}\\ 2^5}\right)\left(\substack{\mathrm{id}\\ 2\cdot 3\cdot 11\tau}\right)\left(\substack{\mathrm{id}\\ 2\cdot 97\tau}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 3^3}\right)$	$(2, 1, 2, 1, 2, 2, 1, 2, 2)$
4	$\left(\substack{\mathrm{id}\\ 2^4\cdot 3\cdot 5}\right)\left(\substack{\mathrm{id}\\ 3\cdot 83}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 13}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 29\tau}\right)\left(\substack{\mathrm{id}\\ 2^6}\right)\left(\substack{\mathrm{id}\\ 2^5}\right)\left(\substack{\mathrm{id}\\ 2\cdot 3\cdot 11\tau}\right)\left(\substack{\mathrm{id}\\ 2\cdot 97\tau}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 3^3}\right)$	$(2, 1, 2, 2, 2, 2, 1, 2, 2)$
3	$\left(\substack{\mathrm{id}\\ 2^4\cdot 3\cdot 5}\right)\left(\substack{\mathrm{id}\\ 3\cdot 83}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 11\tau}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 13}\right)\left(\substack{\mathrm{id}\\ 2^6}\right)\left(\substack{\mathrm{id}\\ 2^5}\right)\left(\substack{\mathrm{id}\\ 2\cdot 3\cdot 11\tau}\right)\left(\substack{\mathrm{id}\\ 2\cdot 97\tau}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 3^3}\right)$	$(2, 1, 2, 1, 2, 2, 1, 2, 2)$
2	$\left(\substack{\mathrm{id}\\ 2^4\cdot 3\cdot 5}\right)\left(\substack{\mathrm{id}\\ 5^3\tau}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 11\tau}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 13}\right)\left(\substack{\mathrm{id}\\ 2^6}\right)\left(\substack{\mathrm{id}\\ 2^5}\right)\left(\substack{\mathrm{id}\\ 2\cdot 3\cdot 11\tau}\right)\left(\substack{\mathrm{id}\\ 2\cdot 97\tau}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 3^3}\right)$	$(2, 1, 2, 1, 2, 2, 1, 2, 2)$
1	$\left(\substack{\mathrm{id}\\ 109\tau}\right)\left(\substack{\mathrm{id}\\ 2^4\cdot 3\cdot 5}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 11\tau}\right)\left(\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 13}\right)\left(\substack{\mathrm{id}\\ 2^6}\right)\left(\substack{\mathrm{id}\\ 2^5}\right)\left(\substack{\mathrm{id}\\ 2\cdot 3\cdot 11\tau}\right)\left(\substack{\mathrm{id}\\ 2\cdot 97\tau}\right)\left(\substack{\mathrm{id}\\ 2^3\cdot 3^3}\right)$	$(1, 2, 2, 1, 2, 2, 1, 2, 2)$

We ended up with our original logarithmic signature as expected and the *vFactorization* vector represents the following element selection / factorization:

$$\left(\boxed{\substack{\mathrm{id}\\ 109\tau}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2^4\cdot 3\cdot 5}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 11\tau}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2^2\cdot 3\cdot 13}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2^6}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2^5}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2\cdot 3\cdot 11\tau}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2\cdot 97\tau}}\right)\left(\boxed{\substack{\mathrm{id}\\ 2^3\cdot 3^3}}\right).$$

Multiplying the selected elements indeed results in σ^{42}, which we wanted to factorize.

9.4.4. Run-Time

Algorithm 9.7 performs $O(n)$ group operations.

- `TauReduce` requires $O(n)$ group operations.

- The code within the "While" loop is executed at most n times, because in each iteration $nPow$ is incremented by one and 2^{n-1} is the maximum power of 2 that can occur (and after processing this power g necessarily is id).
  - Finding the element h can be done in constant time. For this, we once need to build a lookup table, with the power of 2 being the key and the element position the value (similar to bucket-sort). Building the table can be done before the "While" loop and requires $O(n)$ group operations.
  - The rest of the "While" loop requires $O(1)$ group operations.
- The "For" loop at the end clearly requires $O(n)$ group operations.

So, all in all the algorithm requires $O(n) + O(n) + n \cdot (1 + O(1)) + O(n) = O(n)$ group operations.

9.4.5. Generalization for Other Groups

For all $n \in \mathbb{N}$ the dihedral group $D_{2 \cdot 2^n}$ is solvable. $\langle \sigma \rangle$ is a normal subgroup of $D_{2 \cdot 2^n}$ and the factor group $D_{2 \cdot 2^n} / \langle \sigma \rangle$ has order 2, i.e. is isomorphic to $\mathbb{Z}_2$ and thus abelian. The factor group $\langle \sigma \rangle / \{\mathrm{id}\}$ is isomorphic to $\langle \sigma \rangle$, which is cyclic and thus abelian. So, we have the following normal series with abelian factor groups:

$$\{\mathrm{id}\} \trianglelefteq \langle \sigma \rangle \trianglelefteq D_{2 \cdot 2^n}.$$

What Algorithm 9.7 actually does is to transform the given logarithmic signature (of type $(2, 2, \ldots, 2)$) in such a way that the first n blocks generate $\langle \sigma \rangle$ and the last block contains a complete set of coset representatives of $\langle \sigma \rangle$ in $D_{2 \cdot 2^n}$. Deciding which coset representative has to be selected is easy: if the element to factorize is a reflection, select the reflection, otherwise select the rotation.

By rearranging the first n blocks, we could obtain an exact transversal logarithmic signature (in which factoring is easy), where the blocks contain complete sets of coset representatives for the following chain of subgroups:

$$\{\mathrm{id}\} = \left\langle \sigma^{2^n} \right\rangle < \ldots < \left\langle \sigma^4 \right\rangle < \left\langle \sigma^2 \right\rangle < \langle \sigma \rangle .$$

The algorithm does not rearrange the blocks, but in fact it just computes the correct coset representatives in the chain from the right to the left, i.e. it first computes the required coset representative for $\left\langle \sigma^2 \right\rangle$ in $\langle \sigma \rangle$, then for $\left\langle \sigma^4 \right\rangle$ in $\left\langle \sigma^2 \right\rangle$, and so on.

The two block substitutions in Section 9.1.1 when written generically are:

$$\begin{pmatrix} \mathrm{id} \\ g \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ h \end{pmatrix} \leftrightarrow \begin{pmatrix} \mathrm{id} \\ h \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ gh \end{pmatrix}$$
$$\begin{pmatrix} \mathrm{id} \\ g \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ h \end{pmatrix} \leftrightarrow \begin{pmatrix} \mathrm{id} \\ gh \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ h \end{pmatrix}$$

For the first transformation to be valid, it is required that $hgh = g$. In dihedral groups, when g is a reflection and h is a rotation, this indeed is true.

The second transformation is only valid if $ghh = g$, i.e. $h = h^{-1}$. In dihedral groups, reflections indeed have order 2.

Proposition 9.8. *Let $G = D_{2\cdot 2^m} \times \mathbb{Z}_2^k$ with $m, k \in \mathbb{N}_0$, $n := m + k + 1$, and $\alpha \in \Lambda(G)$ canonical with $t(\alpha) = (2, 2, \ldots, 2)$. Then factoring with respect to α is possible using $O(n^2)$ group operations, i.e. α is tame.*

Proof. Observe that the τ-reduction transformation still works fine in G (because all elements in $(\mathbb{Z}_2, +)$ are self-inverse):

$$\begin{pmatrix} \mathrm{id} \\ (\sigma^r\tau, b_1, \ldots, b_k) \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ (\sigma^s, c_1, \ldots, c_k) \end{pmatrix} \leftrightarrow \begin{pmatrix} \mathrm{id} \\ (\sigma^s, c_1, \ldots, c_k) \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ (\sigma^{r-s}\tau, b_1+c_1, \ldots, b_k+c_k) \end{pmatrix},$$
$$\begin{pmatrix} \mathrm{id} \\ (\sigma^r\tau, b_1, \ldots, b_k) \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ (\sigma^s\tau, c_1, \ldots, c_k) \end{pmatrix} \leftrightarrow \begin{pmatrix} \mathrm{id} \\ (\sigma^{r-s}, b_1+c_1, \ldots, b_k+c_k) \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ (\sigma^s\tau, c_1, \ldots, c_k) \end{pmatrix}.$$

So, use the τ-reduction algorithm in G (analogous to the one in Section 9.4.1) to transform α into a logarithmic signature $\alpha' := (A'_1, A'_2, \ldots, A'_{n-1}, B)$ with B being the only block containing an element with a reflection in the first component, and $(A'_1, A'_2, \ldots, A'_{n-1})$ forming a logarithmic signature for $G^\diamond := \mathbb{Z}_{2^m} \oplus \mathbb{Z}_2^k$. With Algorithm 8.5, factoring in $G^\diamond$ can be done using $O(n^2)$ group operations. Consequently factoring in G is easy: when trying to factor an element $g \in G$, select the correct element b in B (if the first component of g is a reflection, the second element in B must be selected, otherwise the first), factor gb^{-1} with respect to α' in $G^\diamond$, and undo the τ-reduction transformation.

In total, we need

$$\underbrace{O(n)}_{\tau\text{-red.}} + \underbrace{O(1)}_{B} + \underbrace{O(n^2)}_{\text{Alg. 8.5}} + \underbrace{O(n)}_{\text{Undo } \tau\text{-red.}} = O(n^2)$$

group operations. □

9.5. Generating $\alpha \in \Lambda(D_{2n})$ with n Odd

Algorithm 9.9. Let $G = D_{2n}$ with n odd, and $v \in \mathbb{P}^k$ a type vector of a logarithmic signature for G (such that $\prod_{i=1}^k v[i] = |G|$).

When passing G and v as input to the following algorithm, it returns a random canonical logarithmic signature $\alpha \in \Lambda(G)$ with $t(\alpha) = v$. By "random" we here mean one logarithmic signature out of the set of all logarithmic signatures that the randomized algorithm can generate, which not necessarily is the set of all logarithmic signatures; we analyze this in Proposition 9.10 and Remark 9.11.

`Random`(m) is expected to generate a uniformly distributed random number between 0 and $m - 1$ (both inclusively). When A is a block of group elements, $|A|$ denotes the length of the block (number of group elements, counting duplicates).

Function `GenDihLogSig`(Group G, TypeVector v) : $\Lambda(G)$

1. Let $n \leftarrow \frac{|G|}{2}$. Set $\alpha \leftarrow (A_1, A_2, \ldots, A_k)$ with $A_i \leftarrow (\underbrace{\mathrm{id}, \mathrm{id}, \ldots, \mathrm{id}}_{v[i]})$ for $1 \leq i \leq k$.
2. Let $x_2 \leftarrow v.\texttt{IndexOf}(2)$. Set $d \leftarrow \texttt{Random}(n)$ and $\alpha[x_2][2] \leftarrow \sigma^d\tau$.
3. Set *xLastTauModInclL* $\leftarrow 1$ and *xLastTauModInclR* $\leftarrow k$.
4. While *xLastTauModInclL* $< x_2$:
   - If $\texttt{Random}(2^{|\alpha[xLastTauModInclL]|-1}) \neq 0$: `break`,
     else: set *xLastTauModInclL* $\leftarrow$ *xLastTauModInclL* $+ 1$.
5. While *xLastTauModInclR* $> x_2$:
   - If $\texttt{Random}(2^{|\alpha[xLastTauModInclR]|-1}) \neq 0$: `break`,
     else: set *xLastTauModInclR* $\leftarrow$ *xLastTauModInclR* $- 1$.
6. Set *lRemBlkIdc* $\leftarrow \{1, 2, \ldots, k\} \setminus \{x_2\}$ and *biGrp* $\leftarrow n$.
7. While *lRemBlkIdc*.`Count` > 0:
   a) Set $x \leftarrow$ *lRemBlkIdc*[$\texttt{Random}$(*lRemBlkIdc*.`Count`) $+ 1$]. Set *blk* $\leftarrow \alpha[x]$ as reference.
   b) Let *cEl* $\leftarrow |$*blk*$|$ and *biSubGen* $\leftarrow \frac{biGrp}{cEl}$.
   c) If $(x \leq$ *xLastTauModInclL*$) \vee (x \geq$ *xLastTauModInclR*$)$:
      - For $1 \leq i <$ *cEl*:
        - Set $s \leftarrow$ (*biSubGen*$\cdot i +$ *biGrp*$\cdot \texttt{Random}(\frac{n}{biGrp}))$ mod n, and *blk*$[i+1] \leftarrow \sigma^s$.

      else:
      i. Set *lSubEl* $\leftarrow \{$*biSubGen* $\cdot\, i \mid i \in \{1, 2, \ldots,$ *cEl* $- 1\}\}$ and $y \leftarrow 2$.
      ii. Set $z \leftarrow$ *biSubGen* $\cdot\, \texttt{Random}($*cEl*$)$.
      iii. If $z \neq 0$:
         A. Set $z \leftarrow (z +$ *biGrp* $\cdot\, \texttt{Random}(\frac{n}{biGrp}))$ mod n.
         B. Set *blk*$[2] \leftarrow \sigma^z$ and *lSubEl* $\leftarrow$ *lSubEl* $\setminus \{z \bmod$ *biGrp*$\}$.
         C. If $2 \mid z$: set *zFixedPoint* $\leftarrow \frac{z}{2}$,
            else: set *zFixedPoint* $\leftarrow (z \cdot \frac{n+1}{2})$ mod n.
         D. Set *blk*$[3] \leftarrow \sigma^{zFixedPoint}$ and *lSubEl* $\leftarrow$ *lSubEl* $\setminus \{$*zFixedPoint* mod *biGrp*$\}$.
         E. Set $y \leftarrow y + 2$.
      iv. While $y \leq$ *cEl*:
         A. Set $u \leftarrow$ *lSubEl*$[1]$.
         B. Set $r \leftarrow (u +$ *biGrp* $\cdot\, \texttt{Random}(\frac{n}{biGrp}))$ mod n.
         C. Set *blk*$[y] \leftarrow \sigma^r$ and *blk*$[y+1] \leftarrow \sigma^{(z-r) \bmod n}$.

D. Set $lSubEl \leftarrow lSubEl \setminus \{u, (z-r) \bmod biGrp\}$.

E. Set $y \leftarrow y + 2$.

d) Set $lRemBlkIdc \leftarrow lRemBlkIdc \setminus \{x\}$ and $biGrp \leftarrow \frac{biGrp}{cEl}$.

8. `MoveAndSelShiftRec`$(G, \alpha, x_2, d, false, xLastTauModInclL)$.

9. `MoveAndSelShiftRec`$(G, \alpha, x_2, d, true, xLastTauModInclR)$.

10. Shuffle all elements except id within the blocks of α randomly.

11. `Return` α.

Function `MoveAndSelShiftRec`(Group G, LogSig α, Int x, Int d, Bool $bRight$, Int $xLastTauModIncl$)

1. If $\overline{bRight} \wedge (x \leq xLastTauModIncl)$: `return`.

2. If $bRight \wedge (x \geq xLastTauModIncl)$: `return`.

3. If $bRight$: set $\widetilde{x} \leftarrow x + 1$,
   else: set $\widetilde{x} \leftarrow x - 1$.

4. Let $S \leftarrow \alpha[\widetilde{x}]$.

5. If $(\overline{bRight} \wedge (\widetilde{x} > xLastTauModIncl)) \vee (bRight \wedge (\widetilde{x} < xLastTauModIncl))$:
   a) Let $Z \leftarrow \{z \in S \mid \{(z-r) \bmod n \mid r \in S\} = S\}$.
   b) Set $z \leftarrow Z[\texttt{Random}(|Z|) + 1]$.
   c) If $bRight$: set $\widetilde{d} \leftarrow (d - z) \bmod n$,
      else: set $\widetilde{d} \leftarrow (d + z) \bmod n$.
   d) `MoveAndSelShiftRec`$(G, \alpha, \widetilde{x}, \widetilde{d}, bRight, xLastTauModIncl)$.

6. For $2 \leq i \leq |\alpha[\widetilde{x}]|$:
   - If `Random`$(2) = 0$:
     - If $bRight$: set $\alpha[\widetilde{x}][i] \leftarrow \sigma^d\tau \cdot \alpha[\widetilde{x}][i]$,
       else: set $\alpha[\widetilde{x}][i] \leftarrow \alpha[\widetilde{x}][i] \cdot \sigma^d\tau$.

Comments.

- The algorithm can roughly be divided into two phases. In the first phase, a logarithmic signature for $\mathbb{Z}_n$ with a few special properties is generated. In the second phase, the reflection element in the block of size 2 is spreaded to the left and to the right using swaps and selective shifts.

- In the beginning, the algorithm locates the position x_2 of the block of size 2 (this position is unique, because n is odd). This block must be of the form $(\mathrm{id}, \sigma^d\tau)$. If it would contain a rotation as second element, it would be impossible for the block sequence to generate exactly n rotations and n reflections. d can be chosen randomly; all values are possible.

- The algorithm then decides up to which blocks the reflection can be spreaded later. These blocks will have indices *xLastTauModInclL* and *xLastTauModInclR*. All blocks with indices $<$ *xLastTauModInclL* or $>$ *xLastTauModInclR* are guaranteed to contain rotations only.

  Determining *xLastTauModInclL* and *xLastTauModInclR* is done probabilistically. When given a random logarithmic signature, we might expect rotations to occur as often as reflections, i.e. an arbitrary element except id in the blocks of α is a rotation with probability $\frac{1}{2}$ and a reflection with probability $\frac{1}{2}$. For example, the first block contains only rotations (excluding id) with probability $\frac{1}{2^{|\alpha[1]|-1}}$, thus the algorithm moves the left boundary *xLastTauModInclL* by one to the right if and only if generating a random number between 0 and $2^{|\alpha[1]|-1}$ (0 inclusive, upper bound exclusive) results in 0. This is iterated for the next blocks. Clearly, x_2 is the maximum. Analogously, the right boundary *xLastTauModInclR* is moved to the left.

- Subsequently, a special logarithmic signature for $\mathbb{Z}_n$ (consisting of all blocks except the one at index x_2) is generated.

  lRemBlkIdc is a list of block indices that we have not processed yet. *biGrp* is the order of the current subgroup, initially being n.

  The algorithm randomly picks one index x from *lRemBlkIdc*. *biSubGen* is chosen such that it generates a subgroup of order *cEl* within the factor group $\mathbb{Z}_n/H$, where H is the subgroup of $\mathbb{Z}_n$ with $|H| = \frac{n}{biGrp}$.

  There are no additional restrictions for the blocks with indices $\leq$ *xLastTauModInclL* and $\geq$ *xLastTauModInclR*.

  All blocks with indices $>$ *xLastTauModInclL* and $<$ *xLastTauModInclR* are additionally required to be size-permutable with the block of size 2 (this is required for the subsequent reflection spreading). Here, we proceed as follows:

  - *lSubEl* is the set of subgroup elements that we have not processed yet. 0 is not in *lSubEl*, because id already is in the block.
  - By Theorem 9.2, a block R (containing only rotations) is size-permutable with the block of size 2 if and only if there exists a $z \in R$ with $\{z - r \mid r \in R\} = R$ (in $\mathbb{Z}_n$).

    So, we pick one of the non-id subgroup elements as z and ensure in the following that the size-permutability condition is fulfilled.

    For this, we need to understand the mapping

    $$\pi_z : \mathbb{Z}_n \to \mathbb{Z}_n : r \mapsto z - r$$

in detail. It is bijective for all $z \in \mathbb{Z}_n$, i.e. π_z is a permutation on $\mathbb{Z}_n$. As $\pi_z(\pi_z(r)) = z - (z - r) = r$, the cycles of π_z have a maximum length of 2. We have $r = z - r \Leftrightarrow 2r = z$; due to $\gcd(2, n) = 1$, this equation has exactly one solution (namely $z \cdot 2^{-1} \bmod n$). So, π_z consists of exactly one fixed point and $\frac{n-1}{2}$ transpositions (cycles of length 2).

- If $z = 0$, z is the fixed point of π_z and there is nothing special to do right now.

  Now assume $z \neq 0$. We put σ^z into the block, thus covering and removing $z \bmod$ *biGrp* from *lSubEl*. 0 and z form a cycle of length 2 for π_z (because we have $\pi_z(0) = z - 0 = z \neq 0$ and $\pi_z(z) = z - z = 0 \neq z$). As π_z has exactly one uniquely determined fixed point, we immediately put it into the block, too. Due to $2^{-1} = \frac{n+1}{2}$ in $\mathbb{Z}_n$, the fixed point is $z \cdot 2^{-1} \bmod n = z \cdot \frac{n+1}{2} \bmod n$.

- For the remaining elements in *lSubEl*, we pick one (it does not matter which one; the algorithm uses the first one) and put a random representative σ^r for it into the block. Size-permutability immediately requires $\sigma^{(z-r) \bmod n}$ to be in the block. The representatives are removed from *lSubEl*. This is iterated until *lSubEl* is empty, i.e. until all elements in the current block have been assigned.

- When the block has been processed, its index is removed from *lRemBlkIdc* and the order *biGrp* of the current factor group is reduced by a factor of *cEl*.

- In the second phase, reflections are spreaded. This is done by two invocations of `MoveAndSelShiftRec`.

  This function first recursively calls itself until ending up at the boundary (moving to the left from x_2 up to *xLastTauModInclL* or moving to the right from x_2 up to *xLastTauModInclR*). On each level one of the possible z is chosen such that the block of size 2 can be swapped with the next block (having index $\widetilde{x}$), and d is updated accordingly (see Theorem 9.2). For performance reasons, we do not actually swap the blocks, but simply update the x and d values.

  When moving up from the recursive chain, on each level we imaginarily undo the swap of the current block and the block of size 2 (except on the bottommost recursion level) and randomly perform selective shifts from the block of size 2 (that now imaginarily is a neighbor of the current block, in the direction to x_2) onto the current block.

- The algorithm description prefers clarity over performance. There are various performance optimizations possible, like for example computing the value $\frac{n}{biGrp}$ once per *lRemBlkIdc* round only and storing it in a variable, rearranging statements to be able to avoid duplicate modulo computations, etc.

  For an optimized (but harder to read) implementation, please see the method `DmDihedralGroup.CreateRandomSig` in the *DmDihedralGroup.Gen.cs* source code file of the LogSig utility (Chapter 12).

Implementation. We have implemented Algorithm 9.9 (with performance optimizations). The LogSig utility (Chapter 12) can be invoked with the command line option "`-gendih`" to generate logarithmic signatures for dihedral groups. The parameter "`-n:`" specifies the order of the group. Using the optional parameter "`-bstype:`", the type of the logarithmic signature can be specified. If "`-sort`" is passed, the elements within the blocks are sorted. If "`-countonly`" is passed, only the number of generated logarithmic signatures is printed (otherwise the logarithmic signatures are printed). Using the optional parameter "`-c:`", the number of logarithmic signatures to generate can be specified (if "`-c:`" is omitted, as many logarithmic signatures as possible are generated).

For example, the following command generates four unique canonical logarithmic signatures α for $D_{2\cdot 15}$ of type $t(\alpha) = (2, 3, 5)$ and the elements within the blocks are sorted:

```
LogSig.exe -gendih -n:2*15 -bstype:2,3,5 -sort -c:4
```

Sample output:

```
D2*15, type: (2, 3, 5)
((id, 4t), (id, 9t, 14t), (id, 8, 2t, 3t, 5t))
((id, 5t), (id, 0t, 10t), (id, 8, 3t, 11t, 14t))
((id, 9t), (id, 5, 10), (id, 3, 11, 2t, 10t))
((id, 14t), (id, 5, 4t), (id, 8, 9, 12t, 13t))
```

Proposition 9.10. *For the following type vectors $t(\alpha)$ (for an $\alpha \in \Lambda(D_{\prod_{s\in t(\alpha)} s})$), Algorithm 9.9 can generate* all *canonical logarithmic signatures of this type for the group:*

$$\big\{(2,3,3),(3,2,3),(3,3,2),\\ (2,3,5),(2,5,3),(3,2,5),(3,5,2),(5,2,3),(5,3,2),\\ (2,5,5),(5,2,5),(5,5,2)\big\}.$$

Proof. We verified this experimentally. First, we counted all canonical logarithmic signatures of the given type (by enumerating them recursively, not randomized; run LogSig with the command line parameters `-enumdih -sort -countonly`, "`-n:`" specifying the group order and "`-bstype:`" specifying the type). Subsequently, we called Algorithm 9.9 over and over again, recording and counting the unique logarithmic signatures that it has generated, and verified that after some time it indeed has output all logarithmic signatures.

Table 9.1 shows the total number of unique logarithmic signatures that both approaches resulted in.

Group	$t(\alpha)$	# Can. Log. Sig.	# Sorted Can. Log. Sig.
$D_{2\cdot 9}$	$(2,3,3)$	10368	2592
	$(3,2,3)$	10368	2592
	$(3,3,2)$	10368	2592
$D_{2\cdot 15}$	$(2,3,5)$	4320000	90000

Group	$t(\alpha)$	# Can. Log. Sig.	# Sorted Can. Log. Sig.
	$(2,5,3)$	3456000	72000
	$(3,2,5)$	4838400	100800
	$(3,5,2)$	3456000	72000
	$(5,2,3)$	4838400	100800
	$(5,3,2)$	4320000	90000
$D_{2\cdot 25}$	$(2,5,5)$	2880000000	5000000
	$(5,2,5)$	4608000000	8000000
	$(5,5,2)$	2880000000	5000000

Table 9.1.: $|\{\alpha \in \Lambda(D_{2n}) \mid \alpha \text{ canonical, specific } t(\alpha)\}|$

Clearly, for $t(\alpha) = (r_1, \ldots, r_k)$ we have # Sorted can. log. sig. $= \frac{\#\text{ Can. log. sig.}}{\prod_{i=1}^{k}(r_i-1)!}$. □

Remark 9.11. For longer types (at least 4 blocks), Algorithm 9.9 cannot generate all canonical logarithmic signatures.

For example, the block sequence

$$\alpha = \begin{pmatrix} \mathrm{id} \\ 5\tau \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ 8 \\ 10 \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ 9 \\ 18 \end{pmatrix} \begin{pmatrix} \mathrm{id} \\ 6 \\ 2\tau \end{pmatrix}$$

is a canonical logarithmic signature for $D_{2\cdot 3^3}$, but it cannot be generated by Algorithm 9.9, because for the reflection to spread from the leftmost block (of size 2) to the rightmost block, the second and third blocks would each need to be size-permutable with the block of size 2, however the second block does not fulfill this ($0 - 8 \bmod 27 = 19 \notin \alpha[2], 8 - 10 \bmod 27 = 25 \notin \alpha[2], 10 - 8 \bmod 27 = 2 \notin \alpha[2]$).

Note though that

$$B := \alpha[2] \cdot \alpha[3] = \{\mathrm{id}, 8, 10, 9, 18, 17, 26, 19, 1\} = \{\mathrm{id}, 1, 8, 9, 10, 17, 18, 19, 26\}$$

is a size-permutable set; we have $B = \{z - b \mid b \in B\}$ for $z \in \{\mathrm{id}, 9, 18\}$. So, if Algorithm 9.9 would be enhanced to additionally check whether multiple blocks together generate a size-permutable set (and possibly swap the reflection block with the multiple blocks at once during the reflection spreading phase), it could generate α.

When implementing this enhancement, a more efficient test whether a set generated by some blocks is size-permutable will be required. The simple approach used in the Examples 9.3 and 9.4 would be insufficient (because the number of generated elements grows exponentially in the number of blocks); the number of blocks would need to be limited (which reduces the number of logarithmic signatures that can be generated).

Proposition 9.12. *Let $G = D_{2n}$ with n odd, $P := \{p \in \mathbb{P} \mid p \,|\, 2n\}$, $\alpha \in \Lambda(D_{2n})$ generated by Algorithm 9.9 and of type $t(\alpha) = (p_1, \ldots, p_k)$ with $p_i \in P$ for all $1 \le i \le k$.*

If $\prod_{p \in P} p$ is polynomial in $\ell(\alpha)$, then α is tame.

This especially implies that all $\alpha \in \Lambda(D_{2n})$ generated by Algorithm 9.9 and of type $t(\alpha) = (p, \ldots, p, 2, p, \ldots, p)$ for some $p \in \mathbb{P}_{\ge 3}$ are tame.

Proof. If we can undo the reflection spreading of the last phase of Algorithm 9.9 (i.e. achieve that only the block of size 2 contains a reflection), factoring is easy by Proposition 9.6 and Algorithm 8.16. So, our only goal is to find an efficient way to undo the reflection spreading (using transformations where the effects on the factorization of an element are only local, like selective shifts and swapping size-permutable blocks).

We only describe undoing the reflection spreading from the block of size 2 to the right; the case to the left works analogously.

Let $d\tau$ be the reflection in the block of size 2, which we call A_i. The first step is to undo the selective shifts performed at the very end of the generation algorithm. For this, perform selective shifts using $d\tau$ onto an element in A_{i+1} if and only if the element is a reflection. This results in A_{i+1} containing only rotations. If all blocks right of A_{i+1} contain rotations only, we are done (with the direction to the right). If not, A_{i+1} must be size-permutable with A_i; in this case swap the blocks (possibly modifying the reflection in A_i) and repeat the procedure recursively.

The only remaining question is which $z \in \mathfrak{Z}(A_j)$ (for any $j \geq i+1$) to choose when swapping A_j with A_i and possibly modifying A_i based on z. One approach is to simply try all possible $z \in \mathfrak{Z}(A_j)$, i.e. assume one of them and continue recursively. At least one of them leads to a solution (i.e. a block sequence that contains rotations only).

Note that each time we encounter a block with multiple possible $z \in \mathfrak{Z}(A_j)$, the possibilities multiply. Thus this approach is only efficient if the total number of possibilities is polynomial in $\ell(\alpha)$. For this, we show that there are at most $\prod_{p \in P} p$ possibilities in total.

By Corollary 2.16 we know exactly how $\mathfrak{Z}(A_j)$ looks like for a block A_j. Let $p := |A_j|$. Either $\mathfrak{Z}(A_j) = \emptyset$ (then A_j is not size-permutable with A_i and we successfully finished this direction), $|\mathfrak{Z}(A_j)| = 1$ (only one possibility for the swap) or $|\mathfrak{Z}(A_j)| = p$. In the last case, we have $E(A_j) = \{\sigma^{i \cdot \frac{n}{p}} \mid 0 \leq i < p\}$ (note this is uniquely determined by p); thus this case can occur only once per prime number in P (except the prime 2, which is irrelevant due to only one block of size 2 occuring).

P contains all the different prime numbers in the type of α. Each of these primes can lead to the $|\mathfrak{Z}(A_j)| \in \mathbb{P}$ situation, where in the worst case all these possibilities multiply. When $\prod_{p \in P} p$ is limited to be polynomial in $\ell(\alpha)$, trying all these possibilities is still efficient. □

9.6. Counting Logarithmic Signatures

9.6.1. $D_{2 \cdot 2^n}$, $t(\alpha) = (2, 2, \ldots, 2)$

By reversing τ-reduction, we can count logarithmic signatures for $D_{2 \cdot 2^n}$ of type $(2, 2, \ldots, 2)$.

Note that we cannot just count the logarithmic signatures having the τ-reduced structure, because the transformation is not injective. For example, the following two logarithmic signatures for $D_{2 \cdot 2}$ are mapped to the same logarithmic signature by a τ-reduction:

$$\binom{\mathrm{id}}{1}\binom{\mathrm{id}}{\tau} \xrightarrow{\tau\text{-Red.}} \binom{\mathrm{id}}{1}\binom{\mathrm{id}}{\tau},$$

$$\binom{\mathrm{id}}{1\tau}\binom{\mathrm{id}}{\tau} \xrightarrow{\tau\text{-Red.}} \binom{\mathrm{id}}{1}\binom{\mathrm{id}}{\tau}.$$

Proposition 9.13. *Let $G = D_{2\cdot 2^n}$. There are*

$$n! \cdot 2^{\frac{n^2+n}{2}} \cdot (2^{n+1} - 1)$$

canonical logarithmic signatures of type $(2, 2, \ldots, 2)$ for G.

Proof. Let $\alpha \in \Xi_{(2,2,\ldots,2)}(G)$ canonical. Consider the following normalization process:

1. Perform a τ-reduction on α.
2. Sort the first n blocks such that $\mathrm{ord}(\alpha[i][2]) \geq \mathrm{ord}(\alpha[i+1][2])$ for all $1 \leq i < n$.
3. Replace $\alpha[i][2]$ by $\min \langle \alpha[i][2] \rangle$ for all $1 \leq i \leq n$.
4. Let $\sigma^k \tau^b \leftarrow \alpha[n+1][2]$. Set $\alpha[n+1][2] \leftarrow \tau^b$.

Every step of this process results in a logarithmic signature if and only if the input was a logarithmic signature.

If $\alpha \in \Lambda(G)$, in step 2 we actually obtain $\mathrm{ord}(\alpha[i][2]) > \mathrm{ord}(\alpha[i+1][2])$ for all $1 \leq i < n$ by Lemma 8.4, and at the end we have $\alpha[n+1][2] = \tau$.

So, $\alpha \in \Lambda(G)$ holds if and only if the process results in

$$\beta := \binom{\mathrm{id}}{2^0}\binom{\mathrm{id}}{2^1}\cdots\binom{\mathrm{id}}{2^{n-1}}\binom{\mathrm{id}}{\tau} \in \Lambda(G).$$

Thus, by starting with β and undoing the process, we can generate all logarithmic signatures of G. In order to determine the number of logarithmic signatures for G, we undo the process and count the number of possibilities in each step.

1. We start with β.

   The reflection in the last block can be translated by any rotation. This gives us 2^n possibilities.

2. We can freely change the coset representatives. The representative σ^1 in the first block can be replaced by $\sigma^3, \sigma^5, \ldots, \sigma^{2^n-1}$ (all odd rotations), i.e. there are 2^{n-1} possibilities for the representative in the first block. For the second block, there are 2^{n-2} possibilities (all rotations being a multiple of 2, but not of 4), for the third there are 2^{n-3} (multiples of 4, but not of 8), etc. So, there are $\prod_{k=0}^{n-1} 2^k = 2^{\sum_{k=0}^{n-1} k} = 2^{\frac{(n-1)n}{2}}$ possibilities.

3. The first n blocks contain rotations only and therefore are freely permutable. This gives us $n!$ possibilities.

4. Now we undo the τ-reduction transformation.

   We start from the right: the last block might contain a reflection or not in the original logarithmic signature, so we need to undo the first or the second τ-reduction selective shift (if the first n blocks of the original logarithmic signature contained a reflection) or stop. For the second-last block the same, etc. However note that we need to stop when reaching the leftmost reflection in the original logarithmic signature. So, there are $\sum_{k=0}^{n} 2^{n-k} = \sum_{k=0}^{n} 2^k = 2^{n+1} - 1$ possibilities (in the first sum the k indicates the zero-based block index of the leftmost reflection and for each k there are $n-k$ blocks right of the kth block, resulting in 2^{n-k} possibilities for each k).

Multiplying all these possibilities, we obtain the total number of canonical logarithmic signatures:

$$\begin{aligned} & 2^n \cdot 2^{\frac{(n-1)n}{2}} \cdot n! \cdot (2^{n+1} - 1) \\ = \; & n! \cdot 2^{\frac{(n-1)n}{2} + n} \cdot (2^{n+1} - 1) \\ = \; & n! \cdot 2^{\frac{n^2+n}{2}} \cdot (2^{n+1} - 1). \end{aligned}$$

□

Results. Table 9.2 shows the number of canonical logarithmic signatures of type $(2, 2, \ldots, 2)$ for the specified groups. For comparison, the number of canonical pseudo-logarithmic signatures (i.e. blocks containing arbitrary elements, but id as first element in each block) is also listed; this number is simply $(2^{n+1})^{n+1}$. The probabilities that were estimated experimentally in Section 6.2 (via relative frequencies) can be computed exactly from Table 9.2 by dividing the number of canonical logarithmic signatures by the number of canonical pseudo-logarithmic signatures.

Group	# Can. Log. Sig.	# Can. Pseudo-Log. Sig.
$D_{2 \cdot 2}$	6	16
$D_{2 \cdot 2^2}$	112	512
$D_{2 \cdot 2^3}$	5760	65536
$D_{2 \cdot 2^4}$	761856	33554432
$D_{2 \cdot 2^5}$	247726080	68719476736
$D_{2 \cdot 2^6}$	191763578880	562949953421312
$D_{2 \cdot 2^7}$	344993248051200	$\approx 1.845 \cdot 10^{19}$
$D_{2 \cdot 2^8}$	$\approx 1.416 \cdot 10^{18}$	$\approx 2.418 \cdot 10^{24}$
$D_{2 \cdot 2^9}$	$\approx 1.306 \cdot 10^{22}$	$\approx 1.268 \cdot 10^{30}$
$D_{2 \cdot 2^{10}}$	$\approx 2.676 \cdot 10^{26}$	$\approx 2.658 \cdot 10^{36}$
$D_{2 \cdot 2^{11}}$	$\approx 1.206 \cdot 10^{31}$	$\approx 2.230 \cdot 10^{43}$
$D_{2 \cdot 2^{12}}$	$\approx 1.186 \cdot 10^{36}$	$\approx 7.482 \cdot 10^{50}$
$D_{2 \cdot 2^{13}}$	$\approx 2.526 \cdot 10^{41}$	$\approx 1.004 \cdot 10^{59}$
$D_{2 \cdot 2^{14}}$	$\approx 1.159 \cdot 10^{47}$	$\approx 5.392 \cdot 10^{67}$
$D_{2 \cdot 2^{15}}$	$\approx 1.139 \cdot 10^{53}$	$\approx 1.158 \cdot 10^{77}$
$D_{2 \cdot 2^{16}}$	$\approx 2.389 \cdot 10^{59}$	$\approx 9.946 \cdot 10^{86}$
$D_{2 \cdot 2^{17}}$	$\approx 1.065 \cdot 10^{66}$	$\approx 3.418 \cdot 10^{97}$

Group	# Can. Log. Sig.	# Can. Pseudo-Log. Sig.
$D_{2\cdot 2^{18}}$	$\approx 1.005 \cdot 10^{73}$	$\approx 4.697 \cdot 10^{108}$
$D_{2\cdot 2^{28}}$	$\approx 2.705 \cdot 10^{160}$	$\approx 1.466 \cdot 10^{253}$

Table 9.2.: $|\{\alpha \in \Lambda(D_{2\cdot 2^n}) \mid \alpha \text{ canonical}, t(\alpha) = (2, 2, \ldots, 2)\}|$

9.6.2. D_{2pq}, $t(\alpha) = (p, 2, q)$

Proposition 9.14. *Let $p, q \in \mathbb{P}_{\geq 3}$, $p \neq q$. Then*

$$\begin{aligned} &|\{\alpha \in \Lambda(D_{2pq}) \mid \alpha \textit{ canonical, } t(\alpha) = (p, 2, q)\}| \\ &= p! \cdot q! \cdot (p^{q-1} + q^{p-1} - 1) \cdot 2^{p+q-2} \end{aligned}$$

and

$$\begin{aligned} &\left|\left\{\alpha \in \Lambda(D_{2p^2}) \mid \alpha \textit{ canonical, } t(\alpha) = (p, 2, p)\right\}\right| \\ &= p!^2 \cdot p^{p-1} \cdot 2^{2p-1}. \end{aligned}$$

Proof. Let $\alpha = (A_1, A_2, A_3) \in \Lambda(D_{2pq})$ canonical with $t(\alpha) = (p, 2, q)$, then we have $A_2 = (\mathrm{id}, d\tau)$ for some $d \in \mathbb{Z}_{pq}$ (the second element in A_2 must be a reflection). By applying selective shifts using $d\tau$ onto the reflections in the surrounding blocks, we can transform α into a standard form $\alpha' = (A_1', A_2', A_3')$, where A_1' and A_3' contain only rotations, and $A_2' = A_2$. Thus in order to count the number of logarithmic signatures, we can first count the standard forms and multiply this by the number of possible selective shifts from A_2.

In a standard form, d is any number in $\mathbb{Z}_{pq}$, i.e. there are pq possibilities.

As A_1' and A_3' contain only rotations and A_2' contains id, we have $(A_1', A_3') \in \Lambda(\mathbb{Z}_{pq})$. By Chapter 8 we know that at least one of the two blocks must be a subgroup and the other one a subgroup modified by selective shifts from the subgroup block. There are $(p-1)!\cdot(q-1)!$ possibilities for both blocks being subgroups (simply arranging the subgroup elements except id). Performing at least one non-id selective shift from A_1' onto A_3', there are $(p-1)!\cdot(q-1)!\cdot(p^{q-1}-1)$ possibilities, because after arranging the subgroup elements within their blocks, onto each element in A_3' except id (there are $q-1$ such elements) one element from A_1' (p possibilities) can be multiplied; 1 possibility must be subtracted, because the $(\mathrm{id}, \ldots, \mathrm{id})$ vector would not result in any change in A_3' and thus we would fall into the previous case. Analogously, for selective shifts from A_3' onto A_1', there are $(p-1)! \cdot (q-1)! \cdot (q^{p-1}-1)$ possibilities. In total we have

$$\begin{aligned} &(p-1)! \cdot (q-1)! \cdot (1 + (p^{q-1} - 1) + (q^{p-1} - 1)) \\ &= (p-1)! \cdot (q-1)! \cdot (p^{q-1} + q^{p-1} - 1) \end{aligned}$$

canonical logarithmic signatures for $\mathbb{Z}_{pq}$ (with two blocks).

Inserting A_2 between (A_1', A_3') results in a logarithmic signature for G (this is correct, because prepending or appending A_2 to (A_1', A_3') clearly results in a logarithmic signature for G and A_2 can be swapped with A_1' or A_3', because one of them is a normal subgroup

in G). Finally, we perform selective shifts from A_2 onto A'_1 and A'_3 to obtain α; there are $2^{p-1} \cdot 2^{q-1} = 2^{p+q-2}$ possibilities for this (for each non-id element in A'_1 and A'_3 either a selective shift is performed or not).

Multiplying all possibilities, we obtain the asserted formula:

$$\begin{aligned} & p \cdot q \cdot (p-1)! \cdot (q-1)! \cdot (p^{q-1} + q^{p-1} - 1) \cdot 2^{p+q-2} \\ = \; & p! \cdot q! \cdot (p^{q-1} + q^{p-1} - 1) \cdot 2^{p+q-2}. \end{aligned}$$

For D_{2p^2} it is almost the same. There are p^2 possibilities for $d\tau$ in A_2. Either A'_1 or A'_3 is a subgroup and the other one is not ($\mathbb{Z}_{p^2}$ is cyclic, not elementary abelian); this gives 2 possibilities. There are p^{p-1} selective shifts from the subgroup block onto the other one. Thus in total we have:

$$\begin{aligned} & p^2 \cdot (p-1)! \cdot (p-1)! \cdot (p^{p-1} + p^{p-1}) \cdot 2^{p+p-2} \\ = \; & p!^2 \cdot p^{p-1} \cdot 2^{2p-1}. \end{aligned}$$

□

Results. For some concrete values p and q, the number of canonical logarithmic signatures α for D_{2pq} of type $t(\alpha) = (p, 2, q)$ is shown in Table 9.3. Additionally, the number of logarithmic signatures when the elements within the blocks are sorted (by any total order on D_{2pq}) is shown; clearly we have

$$\#\text{ Sorted can. log. sig.} = \frac{\#\text{ Can. log. sig.}}{(p-1)! \cdot (q-1)!}.$$

p	q	# Can. Log. Sig.	# Sorted Can. Log. Sig.
3	3	10368	2592
3	5	4838400	100800
3	7	6015098880	4177152
3	11	$\approx 5.804 \cdot 10^{16}$	7997755392
3	13	$\approx 3.254 \cdot 10^{20}$	$\approx 3.397 \cdot 10^{11}$
3	17	$\approx 2.408 \cdot 10^{28}$	$\approx 5.755 \cdot 10^{14}$
3	19	$\approx 2.965 \cdot 10^{32}$	$\approx 2.316 \cdot 10^{16}$
3	23	$\approx 8.166 \cdot 10^{40}$	$\approx 3.633 \cdot 10^{19}$
3	29	$\approx 1.303 \cdot 10^{54}$	$\approx 2.137 \cdot 10^{24}$
5	5	4608000000	8000000
5	7	$\approx 1.116 \cdot 10^{13}$	646016000
5	11	$\approx 7.676 \cdot 10^{20}$	$\approx 8.813 \cdot 10^{12}$
5	13	$\approx 1.196 \cdot 10^{25}$	$\approx 1.040 \cdot 10^{15}$
5	17	$\approx 6.829 \cdot 10^{33}$	$\approx 1.360 \cdot 10^{19}$
7	7	$\approx 2.448 \cdot 10^{16}$	$\approx 4.723 \cdot 10^{10}$
7	11	$\approx 3.748 \cdot 10^{24}$	$\approx 1.434 \cdot 10^{15}$
7	13	$\approx 1.139 \cdot 10^{29}$	$\approx 3.303 \cdot 10^{17}$
7	17	$\approx 2.499 \cdot 10^{38}$	$\approx 1.659 \cdot 10^{22}$
11	11	$\approx 8.667 \cdot 10^{31}$	$\approx 6.582 \cdot 10^{18}$

p	q	# Can. Log. Sig.	# Sorted Can. Log. Sig.
11	13	$\approx 3.416 \cdot 10^{36}$	$\approx 1.965 \cdot 10^{21}$
11	17	$\approx 4.378 \cdot 10^{46}$	$\approx 5.767 \cdot 10^{26}$
13	13	$\approx 3.031 \cdot 10^{40}$	$\approx 1.321 \cdot 10^{23}$
13	17	$\approx 3.960 \cdot 10^{50}$	$\approx 3.951 \cdot 10^{28}$

Table 9.3.: $|\{\alpha \in \Lambda(D_{2pq}) \mid \alpha \text{ canonical}, t(\alpha) = (p, 2, q)\}|$

Remark 9.15. Let $p, q \in \mathbb{P}_{\geq 3}$ and $G = D_{2pq}$. The number of canonical logarithmic signatures for G of type $(2, p, q)$ (and $(p, q, 2)$) is not necessarily the same as for the type $(p, 2, q)$. For examples, see Table 9.1.

10. Other Groups

In Chapter 10, we regard factorizations of other groups.

Our contributions. First we regard generalized quaternion groups, and show that every $\alpha \in \Lambda(Q_{4\cdot 2^n})$ of type $t(\alpha) = (2, 2, \ldots, 2)$ is tame. Building upon this, we prove that every $\alpha \in \Lambda(G)$ of type $t(\alpha) = (2, 2, \ldots, 2)$ is tame for all non-abelian groups of order $|G| = 2^{n+1}$ (with $n \geq 3$) with a cyclic $N \trianglelefteq G$ of order $|N| = 2^n$. For non-abelian groups G of order $|G| = p^{n+1}$ (with $p \in \mathbb{P}_{\geq 3}$ and $n \geq 2$) with a cyclic $N \trianglelefteq G$ of order $|N| = p^n$, $\alpha \in \Lambda(G)$ of type $t(\alpha) = (p, p, \ldots, p)$, we present a reduction to the factorization problem with respect to a p-factorization of $\mathbb{Z}_{p^{n-1}} \oplus \mathbb{Z}_p$, and prove the tameness in a special case.

We then generalize the previous result using homomorphisms with small kernels. As a corollary we show that every $\alpha \in \Lambda(G)$ of type $t(\alpha) = (2, 2, \ldots, 2)$ is tame when G is an extra special 2-group.

Finally, we regard wreath products. We present a few factorization approaches, including an orbit-based factor group descending method.

10.1. Generalized Quaternion Group

Proposition 10.1. *Let $G = Q_{4\cdot 2^n}$ ($n \in \mathbb{N}$), $m := n + 2$, $\alpha = (A_1, \ldots, A_m) \in \Lambda(G)$ canonical of type $(2, 2, \ldots, 2)$, and $A_j = (\mathrm{id}, \sigma^{2^n})$ for a $1 \leq j \leq m$. Then α is tame.*

Proof. Let $A := A_j$. We have $A \trianglelefteq G$, thus the block A can freely be moved to any location in the logarithmic signature without changing the factorization of an element (except permuting the indices order the same of course).

There are two interesting block substitutions:

$$\phi_1 : \begin{pmatrix} \mathrm{id} \\ \sigma^k\tau \end{pmatrix}\begin{pmatrix} \mathrm{id} \\ \sigma^l \end{pmatrix} \leftrightarrow \begin{pmatrix} \mathrm{id} \\ \sigma^l \end{pmatrix}\begin{pmatrix} \mathrm{id} \\ \sigma^{k-l}\tau \end{pmatrix}$$

(generated set in both cases: $\{\mathrm{id}, \sigma^k\tau, \sigma^l, \sigma^{k-l}\tau\}$),

$$\phi_2 : \begin{pmatrix} \mathrm{id} \\ \sigma^k\tau \end{pmatrix}\begin{pmatrix} \mathrm{id} \\ \sigma^l\tau \end{pmatrix}\begin{pmatrix} \mathrm{id} \\ \sigma^{2^n} \end{pmatrix} \leftrightarrow \begin{pmatrix} \mathrm{id} \\ \sigma^{k-l+2^n} \end{pmatrix}\begin{pmatrix} \mathrm{id} \\ \sigma^l\tau \end{pmatrix}\begin{pmatrix} \mathrm{id} \\ \sigma^{2^n} \end{pmatrix}$$

(generated set in both cases: $\{\mathrm{id}, \sigma^k\tau, \sigma^l\tau, \sigma^{k-l+2^n}, \sigma^{2^n}, \sigma^{k+2^n}\tau, \sigma^{l+2^n}\tau, \sigma^{k-l}\}$).

We use these two block substitutions to simplify α (similar to the algorithm in Section 9.4.1):

Function `QuatTauReduce`(LogSig α) : Void

1. For $i \leftarrow 1$ to $m-2$:
   a) Move A to position $i+2$ (without changing the order of the other blocks).
   b) Let $\sigma^k\tau^b \leftarrow \alpha[i][2]$ and $\sigma^l\tau^c \leftarrow \alpha[i+1][2]$.
   c) If $b = 0$: `continue`.
   d) If $c = 0$: apply ϕ_1 to the blocks (A_i, A_{i+1}),
   else apply ϕ_2 to the blocks (A_i, A_{i+1}, A_{i+2}).
2. Move A to position $m-1$.

All of the operations in this algorithm are invertible (so knowing the factorization of an element in the output logarithmic signature, iteratively undoing the block substitutions results in a factorization of the element in the original input logarithmic signature).

In the output logarithmic signature only the last block contains an imaginary element (i.e. an element $\sigma^k\tau^b$ with $b = 1$). The first $m-1$ blocks form a canonical logarithmic signature for the abelian group $\mathbb{Z}_{2^{n+1}}$, in which factoring is possible efficiently, see Chapter 8.

So, in order to factor an element $g \in Q_{4\cdot 2^n}$, perform the simplification above, choose the correct element in the last block (if the element to be factored is imaginary, select the second element, otherwise id), factor the remaining required element in the first $m-1$ blocks, and undo the simplification transformation. □

Example 10.2. Let $G = Q_{4\cdot 2^3}$ and $\alpha = \binom{\mathrm{id}}{\sigma^{11}\tau}\binom{\mathrm{id}}{\sigma^8}\binom{\mathrm{id}}{\sigma^7\tau}\binom{\mathrm{id}}{\sigma^{13}\tau}\binom{\mathrm{id}}{\sigma^{15}} \in \Lambda(G)$. Suppose we want to factor $g = \sigma^5$. The normal subgroup block $A = \binom{\mathrm{id}}{\sigma^8}$ is currently located at position 2.

First we simplify α:

$$\begin{aligned}
&\binom{\mathrm{id}}{\sigma^{11}\tau}\binom{\mathrm{id}}{\sigma^8}\binom{\mathrm{id}}{\sigma^7\tau}\binom{\mathrm{id}}{\sigma^{13}\tau}\binom{\mathrm{id}}{\sigma^{15}}\\
\xrightarrow{\text{Move } A \text{ to position 3}}\ &\binom{\mathrm{id}}{\sigma^{11}\tau}\binom{\mathrm{id}}{\sigma^7\tau}\binom{\mathrm{id}}{\sigma^8}\binom{\mathrm{id}}{\sigma^{13}\tau}\binom{\mathrm{id}}{\sigma^{15}}\\
\xrightarrow{\text{Apply } \phi_2 \text{ to } (A_1, A_2, A_3)}\ &\binom{\mathrm{id}}{\sigma^{12}}\binom{\mathrm{id}}{\sigma^7\tau}\binom{\mathrm{id}}{\sigma^8}\binom{\mathrm{id}}{\sigma^{13}\tau}\binom{\mathrm{id}}{\sigma^{15}}\\
\xrightarrow{\text{Move } A \text{ to position 4}}\ &\binom{\mathrm{id}}{\sigma^{12}}\binom{\mathrm{id}}{\sigma^7\tau}\binom{\mathrm{id}}{\sigma^{13}\tau}\binom{\mathrm{id}}{\sigma^8}\binom{\mathrm{id}}{\sigma^{15}}\\
\xrightarrow{\text{Apply } \phi_2 \text{ to } (A_2, A_3, A_4)}\ &\binom{\mathrm{id}}{\sigma^{12}}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma^{13}\tau}\binom{\mathrm{id}}{\sigma^8}\binom{\mathrm{id}}{\sigma^{15}}\\
\xrightarrow{\text{Move } A \text{ to position 5}}\ &\binom{\mathrm{id}}{\sigma^{12}}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma^{13}\tau}\binom{\mathrm{id}}{\sigma^{15}}\binom{\mathrm{id}}{\sigma^8}\\
\xrightarrow{\text{Apply } \phi_1 \text{ to } (A_3, A_4)}\ &\binom{\mathrm{id}}{\sigma^{12}}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma^{15}}\binom{\mathrm{id}}{\sigma^{14}\tau}\binom{\mathrm{id}}{\sigma^8}\\
\xrightarrow{\text{Move } A \text{ to position 4}}\ &\binom{\mathrm{id}}{\sigma^{12}}\binom{\mathrm{id}}{\sigma^2}\binom{\mathrm{id}}{\sigma^{15}}\binom{\mathrm{id}}{\sigma^8}\binom{\mathrm{id}}{\sigma^{14}\tau}.
\end{aligned}$$

Now factor $g = \sigma^5$ in this simplified logarithmic signature and undo the simplification transformation:

$$\begin{array}{rl}
 & \left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{12}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{2}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{15}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{8}}\end{array}\right)\left(\begin{array}{c}\boxed{\mathrm{id}}\\ \sigma^{14}\tau\end{array}\right)\\
\xrightarrow{\text{Move } A \text{ to position 5}} & \left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{12}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{2}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{15}}\end{array}\right)\left(\begin{array}{c}\boxed{\mathrm{id}}\\ \sigma^{14}\tau\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{8}}\end{array}\right)\\
\xrightarrow{\text{Apply } \phi_1^{-1} \text{ to } (A_3, A_4)} & \left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{12}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{2}}\end{array}\right)\left(\begin{array}{c}\boxed{\mathrm{id}}\\ \sigma^{13}\tau\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{15}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{8}}\end{array}\right)\\
\xrightarrow{\text{Move } A \text{ to position 4}} & \left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{12}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{2}}\end{array}\right)\left(\begin{array}{c}\boxed{\mathrm{id}}\\ \sigma^{13}\tau\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{8}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{15}}\end{array}\right)\\
\xrightarrow{\text{Apply } \phi_2^{-1} \text{ to } (A_2, A_3, A_4)} & \left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{12}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{7}\tau}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{13}\tau}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{8}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{15}}\end{array}\right)\\
\xrightarrow{\text{Move } A \text{ to position 3}} & \left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{12}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{7}\tau}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{8}}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{13}\tau}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{15}}\end{array}\right)\\
\xrightarrow{\text{Apply } \phi_2^{-1} \text{ to } (A_1, A_2, A_3)} & \left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{11}\tau}\end{array}\right)\left(\begin{array}{c}\boxed{\mathrm{id}}\\ \sigma^{7}\tau\end{array}\right)\left(\begin{array}{c}\boxed{\mathrm{id}}\\ \sigma^{8}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{13}\tau}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{15}}\end{array}\right)\\
\xrightarrow{\text{Move } A \text{ to position 2}} & \left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{11}\tau}\end{array}\right)\left(\begin{array}{c}\boxed{\mathrm{id}}\\ \sigma^{8}\end{array}\right)\left(\begin{array}{c}\boxed{\mathrm{id}}\\ \sigma^{7}\tau\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{13}\tau}\end{array}\right)\left(\begin{array}{c}\mathrm{id}\\ \boxed{\sigma^{15}}\end{array}\right).
\end{array}$$

This indeed is the factorization of $g = \sigma^5$ with respect to α.

We now develop an improved factorization approach, which in contrast to Proposition 10.1 does not require the presence of a normal subgroup block.

Proposition 10.3. *Let $G = Q_{4\cdot 2^n}$ $(n \in \mathbb{N})$, $\alpha \in \Lambda(G)$ canonical of type $(2, 2, \ldots, 2)$. Then α is tame.*

Proof. Let $m := n + 2$ (the number of blocks of α), $H := D_{2\cdot 2^n}$ and $\varphi : G \to H : \sigma^k\tau^b \mapsto \sigma^{k \bmod 2^n}\tau^b$ a group homomorphism as in Lemma 2.3. Let $\beta := \varphi(\alpha)$. By Theorem 7.3, β is a 2-factorization of H (i.e. every element of H can be expressed in exactly 2 different ways with respect to β). Now apply the τ-reduction transformation (for dihedral groups, as described in Section 9.4.1) to obtain a block sequence $\gamma \in \Xi(H)$. In γ, only the last block contains a reflection element; denote this reflection element by r. Let ζ be the sequence of the first $m - 1$ blocks of γ, then ζ is a 2-factorization for $\mathbb{Z}_{2^n}$.

Factoring elements $z \in \mathbb{Z}_{2^n}$ with respect to ζ (computing *both* solutions) is possible efficiently. Ignoring any $(0, 0)$ blocks, all other blocks of ζ are cyclic subsets (because they contain 0 and another element not being 0). Thus by Theorem 8.29 at least one of these blocks is a subgroup of $\mathbb{Z}_{2^n}$, and therefore we can apply the factorization Algorithm 8.28 (on all of the first recursion levels until reaching $G = \{0\}$ there exists at least one subgroup block).

Knowing how to compute both factorizations in ζ, we can now describe how to factor in G.

Let $g \in G$. Compute $g' := \varphi(g)$. We are now looking for the two different factorizations of g' with respect to β.

- If g' is a rotation: compute the two factorizations of g' with respect to ζ, select id in the last block of γ and undo the τ-reduction transformation to obtain two solutions in β.
- If g' is a reflection: compute the two factorizations of $g' \cdot r^{-1}$ (where r is the reflection element in the last block of γ, see above) with respect to ζ, select r in the last block of γ and undo the τ-reduction transformation to obtain two solutions in β.

For exactly one of the two solutions, selecting the elements in α with the same indices as in β leads to the factorization of g (and the other solution is the factorization of $\sigma^{2^n} \cdot g$). □

Example 10.4. Let $G = Q_{4 \cdot 2^3}$, $\alpha = \binom{\mathrm{id}}{\sigma^{12}\tau}\binom{\mathrm{id}}{\sigma^8}\binom{\mathrm{id}}{\sigma^{12}}\binom{\mathrm{id}}{\sigma^{13}\tau}\binom{\mathrm{id}}{\sigma^7\tau} \in \Lambda(G)$. Suppose we want to factor $g = \sigma^{11}$. Applying φ, we get

$$\beta = \binom{\mathrm{id}}{\sigma^4\tau}\binom{\mathrm{id}}{\mathrm{id}}\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\sigma^5\tau}\binom{\mathrm{id}}{\sigma^7\tau}$$

as 2-factorization for $D_{2 \cdot 2^3}$. Performing the τ-reduction:

$$\begin{aligned}
\beta &= \binom{\mathrm{id}}{\sigma^4\tau}\binom{\mathrm{id}}{\mathrm{id}}\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\sigma^5\tau}\binom{\mathrm{id}}{\sigma^7\tau} \\
&\to \binom{\mathrm{id}}{\mathrm{id}}\binom{\mathrm{id}}{\sigma^4\tau}\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\sigma^5\tau}\binom{\mathrm{id}}{\sigma^7\tau} \\
&\to \binom{\mathrm{id}}{\mathrm{id}}\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\tau}\binom{\mathrm{id}}{\sigma^5\tau}\binom{\mathrm{id}}{\sigma^7\tau} \\
&\to \binom{\mathrm{id}}{\mathrm{id}}\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\sigma^3}\binom{\mathrm{id}}{\sigma^5\tau}\binom{\mathrm{id}}{\sigma^7\tau} \\
&\to \binom{\mathrm{id}}{\mathrm{id}}\binom{\mathrm{id}}{\sigma^4}\binom{\mathrm{id}}{\sigma^3}\binom{\mathrm{id}}{\sigma^6}\binom{\mathrm{id}}{\sigma^7\tau} =: \gamma,
\end{aligned}$$

and thus

$$\zeta = \binom{0}{0}\binom{0}{4}\binom{0}{3}\binom{0}{6}$$

is the 2-factorization for $\mathbb{Z}_{2^3}$.

As $g' = \varphi(g) = \varphi(\sigma^{11}) = \sigma^3$ is a rotation, id needs to be selected in the last block of γ. The two solutions for 3 in ζ are

$$\binom{\boxed{0}}{0}\binom{\boxed{0}}{4}\binom{0}{\boxed{3}}\binom{\boxed{0}}{6},$$
$$\binom{0}{\boxed{0}}\binom{\boxed{0}}{4}\binom{0}{\boxed{3}}\binom{\boxed{0}}{6}.$$

For both solutions in γ, we now undo the τ-reduction transformation. As first solution we get

$$\begin{aligned}
&\begin{pmatrix}\boxed{\mathrm{id}}\\ \mathrm{id}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^4\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^3}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^6\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix}\\
\to&\begin{pmatrix}\boxed{\mathrm{id}}\\ \mathrm{id}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^4\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^3}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^5\tau\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix}\\
\to&\begin{pmatrix}\boxed{\mathrm{id}}\\ \mathrm{id}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^4\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\tau}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^5\tau}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix}\\
\to&\begin{pmatrix}\boxed{\mathrm{id}}\\ \mathrm{id}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^4\tau}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^4}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^5\tau}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix}\\
\to&\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^4\tau}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \mathrm{id}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^4}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^5\tau}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix}
\end{aligned}$$

and as second solution

$$\begin{aligned}
&\begin{pmatrix}\mathrm{id}\\ \boxed{\mathrm{id}}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^4\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^3}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^6\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix}\\
\to&\begin{pmatrix}\mathrm{id}\\ \boxed{\mathrm{id}}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^4\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^3}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^5\tau\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix}\\
\to&\begin{pmatrix}\mathrm{id}\\ \boxed{\mathrm{id}}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^4\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\tau}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^5\tau}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix}\\
\to&\begin{pmatrix}\mathrm{id}\\ \boxed{\mathrm{id}}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^4\tau}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^4}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^5\tau}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix}\\
\to&\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^4\tau}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\mathrm{id}}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^4}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^5\tau}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix}.
\end{aligned}$$

Selecting the elements at the same indices in α, we obtain the following two factorizations:

$$\begin{aligned}
&\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^{12}\tau}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^8\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^{12}}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^{13}\tau}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix},\\
&\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^{12}\tau}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^8}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^{12}}\end{pmatrix}\begin{pmatrix}\mathrm{id}\\ \boxed{\sigma^{13}\tau}\end{pmatrix}\begin{pmatrix}\boxed{\mathrm{id}}\\ \sigma^7\tau\end{pmatrix}.
\end{aligned}$$

The first factorization generates $\sigma^{12}\tau\cdot\mathrm{id}\cdot\sigma^{12}\cdot\sigma^{13}\tau\cdot\mathrm{id} = \sigma^{-13}\cdot\sigma^8 = \sigma^{-5} = \sigma^{11} = g$ (which we were looking for) and the second factorization generates $\sigma^{12}\tau\cdot\sigma^8\cdot\sigma^{12}\cdot\sigma^{13}\tau\cdot\mathrm{id} = \sigma^{-13} = \sigma^3$ (which indeed is $\sigma^8\cdot g$, where $\sigma^8 \in \ker(\varphi)$).

10.2. Groups G of Order $|G| = p^{n+1}$ with a Cyclic $N \trianglelefteq G$ of Order $|N| = p^n$

Let $n \geq 3$ and G a non-abelian group of order $|G| = 2^{n+1}$ with a cyclic normal subgroup $N = \langle\sigma\rangle \trianglelefteq G$ of order $|N| = 2^n$. Recall that by Section 2.3.5, G is isomorphic to $D_{2\cdot 2^n}$, $Q_{4\cdot 2^{n-1}}$, $\mathfrak{G}_{2\cdot 2^n}$ or $\mathfrak{D}_{2\cdot 2^n}$.

Lemma 10.5. *Let $n \geq 3$ and G a non-abelian group of order $|G| = 2^{n+1}$ with a cyclic $N \trianglelefteq G$ of order $|N| = 2^n$, $\alpha \in \Lambda(G)$ canonical of type $t(\alpha) = (2, 2, \ldots, 2)$. Then α is tame.*

Proof. For the dihedral group $D_{2\cdot 2^n}$, we have shown the tameness in Section 9.4. For the generalized quaternion group $Q_{4\cdot 2^{n-1}}$, we have shown the tameness in Section 10.1.

Define

$$\varphi_{\mathfrak{D}} : \mathfrak{D}_{2\cdot 2^n} \to D_{2\cdot 2^{n-1}} : \sigma^k \tau^b \mapsto \sigma^{k \bmod 2^{n-1}} \tau^b$$

and let $g = \sigma^k \tau^b, h = \sigma^l \tau^c \in \mathfrak{D}_{2\cdot 2^n}$. Then

- If $b = 0$:

$$\begin{aligned} \varphi_{\mathfrak{D}}(g \cdot h) &= \varphi_{\mathfrak{D}}(\sigma^k \cdot \sigma^l \tau^c) = \varphi_{\mathfrak{D}}(\sigma^{k+l} \tau^c) = \sigma^{k+l \bmod 2^{n-1}} \tau^c \\ &= \sigma^{k \bmod 2^{n-1}} \cdot \sigma^{l \bmod 2^{n-1}} \tau^c = \varphi_{\mathfrak{D}}(g) \cdot \varphi_{\mathfrak{D}}(h). \end{aligned}$$

- If $b = 1$:

$$\begin{aligned} \varphi_{\mathfrak{D}}(g \cdot h) &= \varphi_{\mathfrak{D}}(\sigma^k \tau \cdot \sigma^l \tau^c) = \varphi_{\mathfrak{D}}(\sigma^{k-l+l\cdot 2^{n-1}} \tau^{c+1}) \\ &= \sigma^{k-l+l\cdot 2^{n-1} \bmod 2^{n-1}} \tau^{c+1} = \sigma^{k \bmod 2^{n-1}} \cdot \sigma^{-l \bmod 2^{n-1}} \tau^{c+1} \\ &= \sigma^{k \bmod 2^{n-1}} \tau \cdot \sigma^{l \bmod 2^{n-1}} \tau^c = \varphi_{\mathfrak{D}}(g) \cdot \varphi_{\mathfrak{D}}(h). \end{aligned}$$

Thus $\varphi_{\mathfrak{D}}$ is a group homomorphism from $\mathfrak{D}_{2\cdot 2^n}$ to $D_{2\cdot 2^{n-1}}$. With this, we can use the same factorization approach as in Proposition 10.3 (i.e. apply $\varphi_{\mathfrak{D}}$ to obtain a 2-factorization for $D_{2\cdot 2^{n-1}}$, perform a τ-reduction, compute the solutions in $\mathbb{Z}_{2^{n-1}}$, and undo all the transformations).

The group $\mathfrak{G}_{2\cdot 2^n}$ remains. Define

$$\varphi_{\mathfrak{G}} : \mathfrak{G}_{2\cdot 2^n} \to \mathbb{Z}_{2^{n-1}} \oplus \mathbb{Z}_2 : \sigma^k \tau^b \mapsto (k \bmod 2^{n-1}, b)$$

and let $g = \sigma^k \tau^b, h = \sigma^l \tau^c \in \mathfrak{G}_{2\cdot 2^n}$. Then

- If $b = 0$:

$$\begin{aligned} \varphi_{\mathfrak{G}}(g \cdot h) &= \varphi_{\mathfrak{G}}(\sigma^k \cdot \sigma^l \tau^c) = \varphi_{\mathfrak{G}}(\sigma^{k+l} \tau^c) = (k + l \bmod 2^{n-1}, c) \\ &= (k \bmod 2^{n-1}, 0) \cdot (l \bmod 2^{n-1}, c) = \varphi_{\mathfrak{G}}(g) \cdot \varphi_{\mathfrak{G}}(h). \end{aligned}$$

- If $b = 1$:

$$\begin{aligned} \varphi_{\mathfrak{G}}(g \cdot h) &= \varphi_{\mathfrak{G}}(\sigma^k \tau \cdot \sigma^l \tau^c) = \varphi_{\mathfrak{G}}(\sigma^{k+l+l\cdot 2^{n-1}} \tau^{c+1}) \\ &= (k + l + l \cdot 2^{n-1} \bmod 2^{n-1}, c + 1) \\ &= (k \bmod 2^{n-1}, 1) \cdot (l \bmod 2^{n-1}, c) = \varphi_{\mathfrak{G}}(g) \cdot \varphi_{\mathfrak{G}}(h). \end{aligned}$$

Thus $\varphi_{\mathfrak{G}}$ is a group homomorphism from $\mathfrak{G}_{2\cdot 2^n}$ to $\mathbb{Z}_{2^{n-1}} \oplus \mathbb{Z}_2$. In order to factor an element $g \in \mathfrak{G}_{2\cdot 2^n}$, first factor $\varphi_{\mathfrak{G}}(g)$ in the 2-factorization $\varphi_{\mathfrak{G}}(\alpha)$ of $\mathbb{Z}_{2^{n-1}} \oplus \mathbb{Z}_2$ (which is possible efficiently, see the abelian part in the proof of Proposition 10.3), then one of the two factorizations corresponds to the factorization of g in α. □

Proposition 10.6. *Let $p \in \mathbb{P}_{\geq 3}$, $n \geq 2$ and G a non-abelian group of order $|G| = p^{n+1}$ with a cyclic $N \trianglelefteq G$ of order $|N| = p^n$, $\alpha \in \Lambda(G)$ canonical of type $t(\alpha) = (p, p, \ldots, p)$.*

Then factoring with respect to α is as hard as factoring with respect to a p-factorization of $G^\diamond := \mathbb{Z}_{p^{n-1}} \oplus \mathbb{Z}_p$.

If α contains a block A_i with $A_i = \left\langle \sigma^{p^{n-1}} \right\rangle$ or if α contains two blocks A_i, A_j with $A_i \cdot A_j = \left\langle \sigma^{p^{n-1}}, \tau \right\rangle$, then α is tame.

Proof. As seen in Section 2.3.5, $G \cong \mathfrak{G}_{p\cdot p^n} = \left\langle \sigma, \tau \mid \sigma^{p^n} = \tau^p = \mathrm{id}, \tau^{-1}\sigma\tau = \sigma^{1+p^{n-1}} \right\rangle$. Clearly, $\varphi : G \to G^\diamond : \sigma^k\tau^c \mapsto (k \bmod p^{n-1}, c)$ is a group homomorphism (due to the modulo computation, the effect of τ on σ vanishes). Thus, in order to factor a $g \in G$ with respect to α, it is sufficient to find the p different factorizations of $\varphi(g)$ with respect to the p-factorization $\varphi(\alpha)$ of $G^\diamond$.

Let α contain a block A_i with $A_i = \left\langle \sigma^{p^{n-1}} \right\rangle$. We have $\varphi(a) = (0, 0)$ for all $a \in A_i$, thus the multiplicity p in the p-factorization $\varphi(\alpha)$ of $G^\diamond$ is generated completely by the block A_i. So, by removing the block A_i from α and applying φ to the elements in the remaining blocks, we obtain a $\beta \in \Lambda(G^\diamond)$. As $G^\diamond$ is abelian and the blocks of β are all of prime size, β is tame (using Algorithm 8.16). In order to factor a $g \in G$, first factor $\varphi(g)$ with respect to β, choose the same indices in α and complete the factorization by choosing the correct element in A_i (using brute-force, only p possibilities).

Let α contain two blocks A_i, A_j with $A_i \cdot A_j = \left\langle \sigma^{p^{n-1}}, \tau \right\rangle$. Clearly, $\phi : G \to \mathbb{Z}_{p^{n-1}} : \sigma^k\tau^c \mapsto k \bmod p^{n-1}$ is a group homomorphism. $\phi(\alpha)$ is a p^2-factorization of $\mathbb{Z}_{p^{n-1}}$ and $\phi(a) = 0$ for all $a \in A_i \cup A_j$. The factorization of a $g \in G$ can be found like above. □

10.3. Small Kernels and Multiple Factorizations

Theorem 7.4 has shown that factoring in a logarithmic signature α for a group G is possible efficiently when there is a group homomorphism $\varphi : G \to H$ with $k := |\ker(\varphi)|$ being polynomial in $\ell(\alpha)$ and when additionally being able to factor efficiently in the k-factorization $\varphi(\alpha)$ of H. The previous sections used precisely this.

We now state a proposition that generalizes the results from the previous sections (by restricting only H, not G, and not requiring a large normal subgroup of a specific size).

Proposition 10.7. *Let G, H be groups, $\varphi : G \to H$ a surjective group homomorphism, $k := |\ker(\varphi)|$, and $\alpha \in \Lambda(G)$ canonical of type $(2, 2, \ldots, 2)$. If k is polynomial in $\ell(\alpha)$ and any of the following conditions is true, then we can efficiently factor elements with respect to α:*

- *H is cyclic.*
- *H is abelian of type $(2^a, 2, \ldots, 2)$ for some $a \in \mathbb{N}$.*
- *$H = D_{2 \cdot 2^a}$ for some $a \in \mathbb{N}$.*

Proof. First apply φ to α in order to obtain the k-factorization $\varphi(\alpha)$ of H. As α is of type $(2, 2, \ldots, 2)$, the blocks of α are cyclic subsets or $(\mathrm{id}, \mathrm{id})$.

If H is cyclic or abelian of type $(2^a, 2, \ldots, 2)$ for some $a \in \mathbb{N}$, then factoring in $\varphi(\alpha)$ is possible efficiently by using Algorithm 8.28 (on all of the first recursion levels until reaching $|G| = 1$ there exists at least one subgroup block by Theorem 8.29).

If $H = D_{2 \cdot 2^a}$ for some $a \in \mathbb{N}$, then use the approach in the proof of Proposition 10.3.

By Theorem 7.4, α is tame. □

Corollary 10.8. *Let G be an extra special 2-group and $\alpha \in \Lambda(G)$ canonical of type $(2, 2, \ldots, 2)$. Then α is tame.*

Proof. Let $H := G/Z(G)$ and $\varphi : G \to H : g \mapsto g \cdot Z(G)$ the canonical homomorphism from G to H. As G is extra special, $|\ker(\varphi)| = |Z(G)| = 2$ and H is abelian of type $(2, 2, \ldots, 2)$. By Proposition 10.7, α is tame. □

Corollary 10.9. *Let H be an abelian 2-group that is either cyclic or of type $t(H) = (2^a, 2, \ldots, 2)$ with $a \in \mathbb{N}$, or let H be a dihedral 2-group. Let S be an arbitrary group with $|S|$ being polynomial in $\log |H|$.*

Let $G := H \times S$ and $\alpha \in \Lambda(G)$ canonical of type $t(\alpha) = (2, \ldots, 2)$. Then α is tame.

Proof. The projection $\varphi : G \to H : (h, s) \mapsto h$ is a surjective group homomorphism. $|\ker(\varphi)| = |S|$ is polynomial in $\log |H|$ (and thus polynomial in $\ell(\alpha)$). By Proposition 10.7, α is tame. □

10.4. Wreath Products

Proposition 10.10. *Let $G := H \wr_\Omega P$ and $\alpha = (A_1, A_2, \ldots, A_n) \in \Lambda(G)$. Assume that factoring with respect to logarithmic signatures for $H^{|\Omega|}$ and P is possible efficiently. If there exist $l, r \in \mathbb{N}$ such that $U := \bigcup_{l \le i \le r} A_i \subseteq H^{|\Omega|} \times \{\mathrm{id}\}$ and $|U| = |H|^{|\Omega|}$, then α is tame.*

Proof. Let $\varphi : G \to P : ((v_1, v_2, \ldots, v_{|\Omega|}), p) \mapsto p$, $B = (A_l, A_{l+1}, \ldots, A_r)$ and

$$\overline{B} = (A_1, A_2, \ldots, A_{l-1}, A_{r+1}, A_{r+2}, \ldots, A_n).$$

In order to factor a $g \in G$ with respect to α, we can proceed as follows. As $|U| = |H|^{|\Omega|}$, we get $B \in \Lambda(H^{|\Omega|} \times \{\mathrm{id}\})$ and $\varphi(\overline{B}) \in \Lambda(P)$. First find the factorization of $\varphi(g)$ with respect to $\varphi(\overline{B})$ (which is possible efficiently by hypothesis). Select the elements in α at the corresponding positions. Let g_l be the product of the elements selected in the blocks $A_1, A_2, \ldots, A_{l-1}$, and g_r the product of the elements selected in the blocks

$A_{r+1}, A_{r+2}, \ldots, A_n$. Let $g_m := g_l^{-1} g g_r^{-1}$ (with this we have $g = g_l g_m g_r$). We have $g_m \in H^{|\Omega|} \times \{\mathrm{id}\}$. By factoring g_m with respect to B (which is possible efficiently by hypothesis) and selecting the elements in α, we obtain the factorization of g with respect to α. □

Example 10.11. Let $G = \mathbb{Z}_2 \wr \mathbb{Z}_4$ and

$$\alpha = ((\mathrm{id}, ((1,1,0,0),1)), (\mathrm{id}, ((0,1,1,1),0)), (\mathrm{id}, ((1,1,0,0),0)),$$
$$(\mathrm{id}, ((0,0,1,1),0)), (\mathrm{id}, ((1,1,0,1),0)), (\mathrm{id}, ((1,0,1,1),2))) \in \Lambda(G).$$

We want to factor $g = ((1,1,0,0),3)$ with respect to α.

For this, we apply Proposition 10.10. We have $l = 2$ and $r = 5$. The factorization of $\varphi(g) = 3$ with respect to $\varphi(\overline{B})$ is

$$\varphi(\overline{B}) = ((0, \boxed{1}), (0, \boxed{2})).$$

Selecting the corresponding elements in α gives

$$\alpha = ((\mathrm{id}, \boxed{((1,1,0,0),1)}), (\mathrm{id}, ((0,1,1,1),0)), (\mathrm{id}, ((1,1,0,0),0)),$$
$$(\mathrm{id}, ((0,0,1,1),0)), (\mathrm{id}, ((1,1,0,1),0)), (\mathrm{id}, \boxed{((1,0,1,1),2)})).$$

So we have $g_l = ((1,1,0,0),1)$ and $g_r = ((1,0,1,1),2)$. We compute $g_m = g_l^{-1} g g_r^{-1} = ((1,0,0,1),3) \cdot ((1,1,0,0),3) \cdot ((1,1,1,0),2) = ((1,0,1,1),0)$. Factoring g_m with respect to the inner four blocks (which form a logarithmic signature for $\mathbb{Z}_2^4 \times \{\mathrm{id}\}$, which is abelian) is easy:

$$B = ((\mathrm{id}, \boxed{((0,1,1,1),0)}), (\mathrm{id}, \boxed{((1,1,0,0),0)}), (\boxed{\mathrm{id}}, ((0,0,1,1),0)), (\boxed{\mathrm{id}}, ((1,1,0,1),0))).$$

Putting this together with the factorizations for g_l and g_r, we obtain the factorization of g with respect to α:

$$\alpha = ((\mathrm{id}, \boxed{((1,1,0,0),1)}), (\mathrm{id}, \boxed{((0,1,1,1),0)}), (\mathrm{id}, \boxed{((1,1,0,0),0)}),$$
$$(\boxed{\mathrm{id}}, ((0,0,1,1),0)), (\boxed{\mathrm{id}}, ((1,1,0,1),0)), (\mathrm{id}, \boxed{((1,0,1,1),2)})).$$

Proposition 10.12. *Let $\Omega = \{1, 2, \ldots, 2k\}$ for some $k \in \mathbb{N}$. Let P be a permutation group acting on Ω with $|P\omega| = 2$ for all $\omega \in \Omega$ and $|P| = 2^k$ (i.e. $P \cong \mathbb{Z}_2^k$). Let $G := \mathbb{Z}_2 \wr_\Omega P$ and $\alpha = (A_1, A_2, \ldots, A_n) \in \Lambda(G)$ canonical of type $t(\alpha) = (2, 2, \ldots, 2)$.*

Factoring with respect to α is possible in time polynomial in $\sqrt[3]{|G|}$.

Proof. W.l.o.g. assume that the orbits of P on Ω are $\{1,2\}, \{3,4\}, \ldots, \{2k-1, 2k\}$. Let

$$N := \left\langle \bigcup_{i=1}^{k} \{((\delta_{1,i}, \delta_{1,i}, \delta_{2,i}, \delta_{2,i}, \ldots, \delta_{k,i}, \delta_{k,i}), \mathrm{id})\} \right\rangle \trianglelefteq G$$

(where $\delta_{i,j}$ is the Kronecker delta). Let $\varphi : G \to G/N : g \mapsto gN$.

Observe that $N \cong \mathbb{Z}_2^k$ and $G/N \cong (\mathbb{Z}_2 \oplus \mathbb{Z}_2)^k \cong \mathbb{Z}_2^{2k}$; we can go from G to $\mathbb{Z}_2^{2k}$ via

$$\begin{aligned}\varphi' : G \to \mathbb{Z}_2^{2k} : ((v_1, v_2, \ldots, v_{2k}), p) \\ \mapsto (v_1 \oplus v_2, 2 - p(2), v_3 \oplus v_4, 4 - p(4), \ldots, v_{2k-1} \oplus v_{2k}, 2k - p(2k)).\end{aligned}$$

In order to factor a $g \in G$ with respect to α, we can first compute all 2^k different factorizations of $\varphi'(g)$ with respect to the 2^k-factorization $\varphi'(\alpha)$ of $\mathbb{Z}_2^{2k}$ (like in Proposition 10.7). One of these factorizations corresponds to the factorization of g with respect to α; testing all of them is possible in time polynomial in $2^k = (2^3)^{\frac{k}{3}} = 8^{\frac{k}{3}} = \sqrt[3]{|G|}$. □

10.4.1. Orbit-Based Factor Group Descending

Let $G := H \wr_\Omega P$. Write $\Omega = \{\omega_1, \omega_2, \ldots, \omega_{n+m}\}$ (the elements of $H^{|\Omega|}$ shall be indexed by the elements of Ω in this order). Let $\Omega^\circ := \{\omega_{n+1}, \omega_{n+2}, \ldots, \omega_{n+m}\}$ and assume that Ω° forms an orbit, i.e. $P\omega = \Omega^\circ$ for all $\omega \in \Omega^\circ$.

Let $N \trianglelefteq H$ and $T := H^n(H/N)^m$. When the first n copies of H are indexed by $\omega_1, \omega_2, \ldots, \omega_n$ and the m copies of H/N by $\omega_{n+1}, \omega_{n+2}, \ldots, \omega_{n+m}$, P acts on T like on H^{n+m}. As Ω° forms an orbit, H components are permuted with other H components and H/N components are permuted with other H/N components.

With the same automorphism as in the definition of a wreath product, define $G^\diamond := T \rtimes P$. The canonical projection from G to $G^\diamond$ is

$$\begin{aligned}\varphi : G \to G^\diamond : ((h_{\omega_1}, h_{\omega_2}, \ldots, h_{\omega_{n+m}}), p) \\ \mapsto ((h_{\omega_1}, h_{\omega_2}, \ldots, h_{\omega_n}, h_{\omega_{n+1}}N, h_{\omega_{n+2}}N, \ldots, h_{\omega_{n+m}}N), p).\end{aligned}$$

φ is a surjective group homomorphism and we have $z := |\ker(\varphi)| = |N|^m$.

Factoring. Let α be an l-factorization of G. By Theorem 7.4, if z and l are polynomial in $\ell(\alpha)$ and if we can efficiently factor elements in the $(z \cdot l)$-factorization $\varphi(\alpha)$ of $G^\diamond$, then we can efficiently factor elements with respect to α.

11. Black Box Groups

In Chapter 11, we regard black box groups. In all previous chapters we assumed that the groups were given in a specific representation (e.g. in Chapter 8, we assumed the abelian group G to be represented as $G = \mathbb{Z}_{p_1^{k_1}} \oplus \mathbb{Z}_{p_2^{k_2}} \oplus \ldots \oplus \mathbb{Z}_{p_m^{k_m}}$ with $p_i \in \mathbb{P}$ and $k_i \in \mathbb{N}$ for $1 \leq i \leq m$), and the algorithms in these chapters usually depend on these specific representations. In Chapter 11, we assume that a group is given as a black box (i.e. any arbitrary representation), supporting only a few basic group operations. Given such a black box group, our goal is to map elements to the representations that we require in the previous chapters. If we succeed, we have shown that the results from the previous chapters actually hold for arbitrary representations of a group with the specific structure.

Our contributions. Our first result is that cyclic black box groups indeed can be mapped to our usual representation $\mathbb{Z}_n$ (integers mod n). For elementary abelian p-groups, we show a few results for the case when an efficient linear dependence test is available. A few weaker results are presented for the general case of an abelian group. For dihedral groups, we show that the mapping is possible efficiently. Furthermore, we show that every $\alpha \in \Lambda(G)$ of type $t(\alpha) = (2, 2, \ldots, 2)$ is tame when G is a non-abelian group (given as black box group) of order $|G| = 2^{n+1}$ having a cyclic normal subgroup $N \trianglelefteq G$ of order $|N| = 2^n$.

11.1. Definition and Fundamental Properties

The *black box group model*, as introduced in [Bab84], is one of the most general models to describe a group:

- Group elements are encoded by binary strings of uniform length b_G (the *code length*). Not every $x \in \mathbb{Z}_2^{b_G}$ needs to represent a group element, and a group element may be represented by multiple different strings.
- There are three oracle functions computed by a black box:
  - $\texttt{prod} : \mathbb{Z}_2^{b_G} \times \mathbb{Z}_2^{b_G} \to \mathbb{Z}_2^{b_G}$ computes the product of two group elements.
  - $\texttt{inv} : \mathbb{Z}_2^{b_G} \to \mathbb{Z}_2^{b_G}$ inverts a group element.
  - $\texttt{id} : \mathbb{Z}_2^{b_G} \times \mathbb{Z}_2^{b_G^c} \to \{\text{YES}, *\}$ (with the *witness exponent* $c \in \mathbb{N}$) with the following properties: if $x \in \mathbb{Z}_2^{b_G}$ represents the identity, then there exists an $y \in \mathbb{Z}_2^{b_G^c}$ such that $\texttt{id}(x, y) = \text{YES}$ (y is called a *witness* of x representing the identity); if $x \in \mathbb{Z}_2^{b_G}$ represents a group element different from the identity, then $\texttt{id}(x, y) = *$ (for all $y \in \mathbb{Z}_2^{b_G^c}$).

These functions are not defined for input strings that do not encode group elements.

A *black box group* is given by the black box (which allows computing the group operations above) and a set of generator elements (by binary strings).

The question whether a string x represents the identity can be answered in NP time: by guessing y we can verify in polynomial time that $\texttt{id}(x, y) = \text{YES}$. However, to answer whether x does not represent the identity would require an exhaustive search of all possible witnesses y. The definition of the function `id` was motivated by factor groups: in order to test whether a string represents the identity in a factor group G/N, the membership of the element in N must be tested, which can be hard.

In the *unique encoding black box group* model, a group element is encoded by exactly one string. This has various implications, the most important one being that identity testing is now possible efficiently: first find the encoding of the identity element (by taking an arbitrary generator element g, computing g^{-1} and $g^{-1}g$), subsequently the identity test for an element x can be realized by comparing the string of x with the string of $g^{-1}g$.

In the following we want to be able to efficiently decide whether a string represents the identity or not. Furthermore, we are primarily interested in (multiple) factorizations of groups. This motivates Definition 11.1.

Definition 11.1. The set $\mathcal{B}$ is defined as follows. A family $(G_n, \alpha_n)_{n\in\mathbb{N}}$ is an element of $\mathcal{B}$, if and only if for all $n \in \mathbb{N}$:

- G_n is a black box group. The identity test oracle function shall be computable efficiently. The order $|G_n|$ shall be known. Furthermore, $|G_n| \leq |G_{n+1}|$ shall hold.
- $\alpha_n = (A_{n,1}, A_{n,2}, \ldots, A_{n,m_n})$ is a (multiple) factorization of G_n (of multiplicity $\frac{\prod_{i=1}^{m_n} |A_{n,i}|}{|G_n|}$). The elements in the blocks of α_n are given by binary strings.

As α_n is a (multiple) factorization of G_n, $E(\alpha_n)$ is a set of generator elements for G_n. We do not require any additional/separate set of generators. Note that $|E(\alpha_n)| \leq \ell(\alpha_n)$.

It is tempting to drop the requirement that $|G_n|$ shall be known, because α_n is available and $|G_n| \mid \prod_{i=1}^{m_n} |A_{n,i}|$. When $\alpha_n \in \Lambda(G_n)$, then $|G_n| = \prod_{i=1}^{m_n} |A_{n,i}|$. However, when α_n is a multiple factorization of multiplicity ≥ 2, it is not immediately clear how to determine the group's order efficiently by inspecting G_n (as black box) and α_n. For example, if G_n is an elementary abelian group, obstacles similar to the ones in Section 11.3 arise.

Notation. Like in Section 3.3, we omit the term "family of" and write $(G, \alpha) \in \mathcal{B}$ instead of $(G_n, \alpha_n)_{n\in\mathbb{N}} \in \mathcal{B}$. For example, by "let $(G, \alpha) \in \mathcal{B}$ with G abelian" we mean to take any family $(G_n, \alpha_n)_{n\in\mathbb{N}} \in \mathcal{B}$ where all G_n are abelian.

When we write that an element $g \in G$ is given, then g is usually given as a binary string.

We continue to use additive notation when knowing that the group is abelian. A "+" for group elements can be realized by a `prod` call of the black box, and a "−" by an `inv`

call. In tests for a specific group (e.g. Proposition 11.4), multiplicative notation is used, because arbitrary (abelian and non-abelian) groups are allowed as input.

Run-time. Many problems on abelian black box groups can be solved using algorithms with a worst-case time complexity polynomial in $|G|$. For example, in [Ili85] algorithms are presented for computing a set of defining relations, a complete basis and the intersection of two abelian groups. In [Buc05], an algorithm is presented that computes the structure of an abelian group G from a generating set M; the algorithm performs $O(|M| \cdot \sqrt{|G|})$ group operations and stores $O(\sqrt{|G|})$ group elements.

However, in the following we are interested in more efficient algorithms. As defined in Section 3.3, the space required to encode a pair (G, α) can be estimated by $\mathcal{S}(G, \alpha) := b_G \cdot \ell(\alpha)$ (where b_G is the code length). We assume the black box realizing G is given, and only a negligible amount of memory is required for identifying/linking to it. Furthermore, we ignore the memory required for storing the order $|G|$, because $\log_2 |G| \leq b_G$, i.e. the space of one group element is sufficient for storing the order $|G|$ and thus the required memory is negligible.

A Las Vegas algorithm having only (G, α) as input is efficient, if the expected value of its run-time is bounded by a polynomial in $\mathcal{S}(G, \alpha)$.

Generators. When a black box group with a code length b_G and s generators is given, the number of required generators can be reduced to $O(b_G)$ using a Monte Carlo algorithm performing $O(s \cdot \log b_G)$ group operations (see Theorem 1.5 in [Bab95]).

Lemma 11.2. *Let $(G, \alpha) \in \mathcal{B}$ and $g, h \in G$. Testing whether $g = h$ is possible efficiently.*

Proof. $g = h \Leftrightarrow g \cdot h^{-1} = \mathrm{id}$; verifying the right side only requires efficient operations/tests. □

Proposition 11.3. *Let $(G, \alpha) \in \mathcal{B}$. When only applying algorithms that perform a number of black box queries polynomial in $\mathcal{S}(G, \alpha)$, then the model is equivalent to a unique encoding black box group model.*

Proof. We construct an equivalent unique encoding black box group $G^\diamond$, such that the algorithms use binary strings of $G^\diamond$, and $G^\diamond$ internally performs computations with elements of G.

The identity test operation of $G^\diamond$ is defined to be exactly the same as in G, i.e. the black box $G^\diamond$ internally just performs the identity test operation of G and returns the result to the caller.

Let us introduce a list L of strings. Initially L is empty. For every element $x \in E(\alpha)$ check whether x is equivalent to an element already in L (using Lemma 11.2); if no equivalent element exists in L: add x to L.

The other two operations (`prod` and `inv`, both returning group elements) are now defined as follows: taking the input parameters, compute the result of the operation in G, but instead of directly returning it to the caller, first check whether the result is equivalent to any element in L; if yes, return the previous string stored in L, otherwise add the string to

L and return it. By this, different string representations of a group element are mapped to the first one encountered, thus $G^\diamond$ is a unique encoding black box.

As the caller algorithms only perform a number of black box queries polynomial in $\mathcal{S}(G,\alpha)$, L also only contains polynomially many elements and searching L on each group operation requires polynomial time. Consequently the realization of $G^\diamond$ based on G requires only polynomial time. □

Proposition 11.4. *Let $(G,\alpha) \in \mathcal{B}$. Then it can be tested efficiently whether G is abelian.*

Proof. For all $a, b \in E(\alpha)$ test whether $a \cdot b = b \cdot a$. At most $|E(\alpha)|^2$ tests are required, and $|E(\alpha)|^2 \leq \ell(\alpha)^2 \leq \mathcal{S}(G,\alpha)^2$ (which is polynomial in $\mathcal{S}(G,\alpha)$). □

Lemma 11.5. *Let $(G,\alpha) \in \mathcal{B}$. $|G|$ can be decomposed into its prime factors $|G| = p_1^{e_1} \cdots p_s^{e_s}$ (with $p_i \in \mathbb{P}$ pairwise different and $e_i \in \mathbb{N}$) efficiently, i.e. in time polynomial in $\mathcal{S}(G,\alpha)$.*

Proof. Given $\alpha = (A_1, A_2, \ldots, A_m)$, compute $M := \prod_{i=1}^m |A_i|$. The prime factorization $M = q_1^{k_1} \cdots q_t^{k_t}$ (with $q_i \in \mathbb{P}$ pairwise different and $k_i \in \mathbb{N}$) can be computed within time polynomial in $\ell(\alpha)$ (factoring each block size $|A_i|$ using trial division is sufficient).

As α is a multiple factorization of G, we have $|G| \mid M$. We can now decrease the exponents k_i of the prime factors q_i of M until obtaining $|G|$ (i.e. decrease each k_i as long as the resulting number is still divisible by $|G|$). This procedure requires at most $\sum_{i=1}^t k_i$ steps, and we have $\sum_{i=1}^t k_i \leq \ell(\alpha) \leq \mathcal{S}(G,\alpha)$. □

Note that in Lemma 11.5 we are measuring in $\mathcal{S}(G,\alpha)$, not $\log_2 |G|$ (which is the input length for integer factorization algorithms).

For details on the connection between the group order and the minimal size of the input (G,α), see Section 3.3.

Lemma 11.6. *Let $(G,\alpha) \in \mathcal{B}$ and $g \in G$. Then* $\mathrm{ord}(g)$ *can be computed efficiently, using $O((\log_2 |G|)^2)$ group operations.*

Proof. We know the order $|G| = p_1^{e_1} \cdots p_s^{e_s}$ (with $p_i \in \mathbb{P}$ pairwise different and $e_i \in \mathbb{N}$) of G (by applying Lemma 11.5). Clearly $g^{|G|} = \mathrm{id}$. The idea now is to remove as many prime factors from $|G|$ as possible (as long as the exponentiation of g still gives id). The minimum remaining exponent then is the order of g.

In detail, the following algorithm computes $\mathrm{ord}(g)$:

1. Set $r \leftarrow p_1^{e_1} \cdots p_s^{e_s}$.
2. For each p_i in $|G|$:
   a) Let $m \leftarrow \max\{t \in \mathbb{N}_0 \mid t \leq e_i \text{ and } g^{\frac{r}{p_i^t}} = \mathrm{id}\}$.
   b) Set $r \leftarrow \frac{r}{p_i^m}$.

At the end, $r = \operatorname{ord}(g)$.

By iterated squaring and multiplying (binary exponentiation), every exponentiation of g can be computed using $O(\log_2 |G|)$ group element multiplications.

$|G|$ contains s pairwise different prime factors. The computation of m can be realized as a loop, starting with $t = e_i$ and decreasing t until the exponentiation of g gives id; we need at most e_i steps for this (in the case $t = 0$ there is no computation required).

So, in total we need

$$\begin{aligned}\sum_{i=1}^{s}\sum_{t=1}^{e_i} O(\log_2 |G|) &= O(\log_2 |G|) \cdot \sum_{i=1}^{s} e_i = O(\log_2 |G|) \cdot \sum_{i=1}^{s} \log_{p_i} p_i^{e_i} \\ &\subseteq O(\log_2 |G|) \cdot \sum_{i=1}^{s} \log_2 p_i^{e_i} = O(\log_2 |G|) \cdot \log_2 \prod_{i=1}^{s} p_i^{e_i} \\ &= O(\log_2 |G|) \cdot \log_2 |G| = O((\log_2 |G|)^2)\end{aligned}$$

group operations. □

11.2. Cyclic Groups

Note that cyclic groups are abelian, thus we use additive notation in this section.

Theorem 11.7. *Let $(G, \alpha) \in \mathcal{B}$ with G a cyclic group. Finding some $g \in G$ with $\langle g \rangle = G$ is possible efficiently.*

Proof. Let $|G| = p_1^{e_1} \cdots p_s^{e_s}$ with $p_i \in \mathbb{P}$ pairwise different and $e_i \in \mathbb{N}$.

For all $1 \leq i \leq s$ define $\widehat{S}_{p_i} := \{\frac{|G|}{p_i^{e_i}} \cdot h \mid h \in E(\alpha)\}$ (then $\left\langle \widehat{S}_{p_i} \right\rangle = \operatorname{Syl}_{p_i}(G)$), $m_i := \max\{\operatorname{ord}(h) \mid h \in \widehat{S}_{p_i}\}$ and $M_i := \{h \in \widehat{S}_{p_i} \mid \operatorname{ord}(h) = m_i\}$. For each i pick one arbitrary $h_i \in M_i$ (observe that $\langle h_i \rangle = \operatorname{Syl}_{p_i}(G)$) and compute $g := \sum_{i=1}^{s} h_i$. With this g, we have $\langle g \rangle = G$, because $\operatorname{ord}(g) = |G|$ due to $\gcd(\operatorname{ord}(h_i), \operatorname{ord}(h_j)) = 1$ for $i \neq j$. □

Theorem 11.8. *Let $(G, \alpha) \in \mathcal{B}$ with $G = \langle g \rangle$ a cyclic group and $h \in G$. Finding a $k \in \mathbb{N}_0$ such that $k \cdot g = h$ is possible efficiently.*

Proof. k is computed by the following algorithm:

1. Set $k \leftarrow 0$ and $r \leftarrow h$.
2. While $r \neq 0$:
   a) Compute $u \leftarrow \operatorname{ord}(r)$ and set $v \leftarrow \frac{|G|}{u}$.
   b) Find the minimal $i \in \mathbb{N}$ for which $\operatorname{ord}(r - i \cdot (v \cdot g)) < u$.
   c) Set $k \leftarrow k + v \cdot i$ and $r \leftarrow r - i \cdot (v \cdot g)$.

The group element sums can be computed efficiently using iterated doubling and adding (in multiplicative notation this is squaring and multiplying, also known as binary exponentiation). Note that $-i \cdot (v \cdot g) = (v \cdot i) \cdot (-g)$.

Let $|G| = p_1^{e_1} \cdots p_s^{e_s}$ with $p_j \in \mathbb{P}$ pairwise different and $e_j \in \mathbb{N}$. Finding i will be implemented using a loop; let us call this loop the inner loop. For determining the runtime of the algorithm, it is sufficient to count how often the inner loop is executed in total. Observe that within each "While" loop $\langle r \rangle = \langle v \cdot g \rangle$ holds. Thus $i \leq \max\{p_j \mid 1 \leq j \leq s\}$, so $\sum_{t=1}^{s} e_t \cdot \max\{p_j \mid 1 \leq j \leq s\}$ is an upper bound for the total maximum number of iterations of the inner loop. However this bound can be improved: in each "While" loop the maximum prime factor in $\operatorname{ord}(r)$ is an upper bound for i, and each "While" loop execution removes at least one prime factor from $\operatorname{ord}(r)$. Thus, summing over all "While" loops, $e_1 p_1 + \ldots + e_s p_s$ is an upper bound for the total maximum number of iterations of the inner loop. Clearly, this is polynomial in $\mathcal{S}(G, \alpha)$. □

Example 11.9. Let $G = \mathbb{Z}_{2^3} \oplus \mathbb{Z}_{3^2} \oplus \mathbb{Z}_5$ (which is cyclic, $G \cong \mathbb{Z}_{2^3 \cdot 3^2 \cdot 5}$), $g = (5, 2, 3)$ (we have $\langle g \rangle = G$ due to $\gcd(5, 2^3) = 1$, $\gcd(2, 3^2) = 1$ and $\gcd(3, 5) = 1$), and $h = (1, 1, 2)$. We want to compute a $k \in \mathbb{N}_0$ such that $k \cdot g = h$ using the algorithm in the proof of Theorem 11.8, while treating G as a black box group and g, h as binary strings. In the following, all computations in parentheses are just done to allow us computing the orders; these representations are unknown to the algorithm.

Set $k \leftarrow 0$ and $r \leftarrow h$.

The black box tells us that $r \neq (0, 0, 0)$, thus we enter the "While" loop. $u \leftarrow \operatorname{ord}(r) = 2^3 \cdot 3^2 \cdot 5$ (using Lemma 11.6 for the computation), $v \leftarrow 1$. The inner loop begins. $\operatorname{ord}(r - (v \cdot g)) = 2 \cdot 3^2 \cdot 5 < u$ (we have $r - (v \cdot g) = (1, 1, 2) - (5, 2, 3) = (4, 8, 4)$), so $i = 1$. Consequently we set $k \leftarrow k + v \cdot i = 0 + 1 \cdot 1 = 1$ and $r \leftarrow r - i \cdot (v \cdot g)$ (so now $r = (4, 8, 4)$).

Again $r \neq (0, 0, 0)$, thus we enter the "While" loop another time. $u \leftarrow \operatorname{ord}(r) = 2 \cdot 3^2 \cdot 5$, $v \leftarrow 2^2 = 4$. $\operatorname{ord}(r - (v \cdot g)) = 5 < u$ (we have $r - (v \cdot g) = (4, 8, 4) - (4 \cdot 5, 4 \cdot 2, 4 \cdot 3) = (0, 0, 2)$), so $i = 1$. Consequently we set $k \leftarrow k + v \cdot i = 1 + 4 \cdot 1 = 5$ and $r \leftarrow r - i \cdot (v \cdot g)$ (so now $r = (0, 0, 2)$).

We get $r \neq (0, 0, 0)$, so another round. $u \leftarrow \operatorname{ord}(r) = 5$, $v = 2^3 \cdot 3^2 = 72$. $\operatorname{ord}(r - (v \cdot g)) = 5 = u$ (we have $r - (v \cdot g) = (0, 0, 2) - (72 \cdot 5, 72 \cdot 2, 72 \cdot 3) = (0, 0, 1)$), $\operatorname{ord}(r - 2 \cdot (v \cdot g)) = 1 < u$ (we have $r - 2 \cdot (v \cdot g) = (0, 0, 2) - (2 \cdot 72 \cdot 5, 2 \cdot 72 \cdot 2, 2 \cdot 72 \cdot 3) = (0, 0, 0)$), so $i = 2$. Consequently we set $k \leftarrow k + v \cdot i = 5 + 72 \cdot 2 = 149$ and $r \leftarrow r - i \cdot (v \cdot g)$ (so now $r = (0, 0, 0)$).

As $r = (0, 0, 0)$, the algorithm terminates. We can easily verify that indeed $k \cdot g = 149 \cdot (5, 2, 3) = (149 \cdot 5, 149 \cdot 2, 149 \cdot 3) = (1, 1, 2) = h$.

Factoring. Let $(G, \alpha) \in \mathcal{B}$ with G a cyclic group, and $g \in G$. Factoring g with respect to α is as hard as in Chapter 8: first find a generator $h \in G$ for G using Theorem 11.7, map all elements in the blocks of α and g to multiples of h using Theorem 11.8, and apply the algorithms in Chapter 8.

Theorem 11.10. *Let G be a cyclic black box group, $B \subseteq G$ with $\langle B \rangle = G$, and $|G| = p_1^{e_1} \cdots p_s^{e_s}$ (with $p_i \in \mathbb{P}$ pairwise different and $e_i \in \mathbb{N}$).*

A canonical $\alpha \in \Lambda(G)$ with minimal length can be constructed efficiently.

Proof. For all $1 \leq i \leq s$ define $\widehat{S}_{p_i} := \{\frac{|G|}{p_i^{e_i}} \cdot h \mid h \in B\}$ (then $\left\langle \widehat{S}_{p_i} \right\rangle = \mathrm{Syl}_{p_i}(G)$), $m_i := \max\{\mathrm{ord}(h) \mid h \in \widehat{S}_{p_i}\}$ and $M_i := \{h \in \widehat{S}_{p_i} \mid \mathrm{ord}(h) = m_i\}$. For each i pick one arbitrary $h_i \in M_i$ (observe that $\langle h_i \rangle = \mathrm{Syl}_{p_i}(G)$). Compute blocks

$$A_i^{(j)} := (0, (1 \cdot p_i^j) \cdot h_i, (2 \cdot p_i^j) \cdot h_i, \ldots, ((p_i - 1) \cdot p_i^j) \cdot h_i)$$

for all $1 \leq i \leq s$ and $0 \leq j \leq e_i - 1$. Let

$$\alpha := (A_1^{(0)}, A_1^{(1)}, \ldots, A_1^{(e_1-1)}, A_2^{(0)}, \ldots, A_2^{(e_2-1)}, \ldots, A_s^{(0)}, \ldots, A_s^{(e_s-1)}).$$

Clearly, $\alpha \in \Lambda(G)$ and it is canonical. As α contains only blocks of prime size, its length $\ell(\alpha) = \sum_{i=1}^{s} e_i p_i$ is minimal. □

In order to emphasize the connection between the group order and the input (G, α) again, we prove a well-known cryptographic result (which is usually proven using the Pohlig-Hellman algorithm [Poh78]):

Corollary 11.11. *Let $G = \langle g \rangle$ a cyclic black box group with $|G|$ containing only prime factors less than $\log_2 |G|$. Then discrete logarithms can be computed efficiently in G, and thus for example Diffie-Hellman key exchange on G is insecure.*

Proof. Use Theorem 11.10 to construct an $\alpha \in \Lambda(G)$ from $\{g\}$. As $|G|$ contains only prime factors less than $\log_2 |G|$, each block of α has a size less than $\log_2 |G|$. Furthermore, α can consist of at most $\log_2 |G|$ blocks. Thus, $\ell(\alpha) < (\log_2 |G|)^2$, i.e. $\ell(\alpha)$ is polynomial in $\log_2 |G|$. Using Theorem 11.8 we efficiently find the discrete logarithm.

If $|G|$ would contain a large prime factor, α would contain a large block, resulting in a higher run-time. □

11.3. Elementary Abelian p-Groups

When G is an elementary abelian p-group, G is isomorphic to a $(\log_p |G|)$-dimensional vector space over the field $\mathbb{Z}_p$.

Proposition 11.12. *Let $(G, \alpha) \in \mathcal{B}$. Then testing whether $(G, \cdot)$ is an elementary abelian p-group is possible efficiently.*

Proof. By the factorization of $|G|$ we get p (or can immediately decide that G is not a p-group when $|G|$ is not a power of p). Now test whether $a \cdot b = b \cdot a$ for all $a, b \in E(\alpha)$ and whether $\mathrm{ord}(g) \in \{1, p\}$ for all $g \in E(\alpha)$. □

Proposition 11.13. *Let $(G, \alpha) \in \mathcal{B}$ with G an elementary abelian p-group. Let $T : \mathcal{P}(G) \to \{\mathrm{true}, \mathrm{false}\}$ a function that efficiently tests whether a set of group elements is linear dependent (in time polynomial in the input length).*

Then finding an independent generating set for G is possible efficiently.

Proof. The key idea is that the linear dependence test provides an efficient subgroup membership test. Let S be a set of linear independent elements, then an element $g \in G$ is in $\langle S \rangle$ if and only if $g \in S$ or $S \cup \{g\}$ is a linear dependent set (which can be tested efficiently using T).

The independent generating set S for G is constructed by the following simple algorithm:

1. Set $S \leftarrow \emptyset$.
2. For each $g \in E(\alpha)$:
   - If not $T(S \cup \{g\})$: set $S \leftarrow S \cup \{g\}$.

The loop is iterated $|E(\alpha)| \leq \ell(\alpha)$ times. An upper bound for the length of the input for T is $b_G \cdot \ell(\alpha)$. Thus, all in all the worst-case run-time of the algorithm is polynomial in $b_G \cdot \ell(\alpha)^2 \leq \mathcal{S}(G, \alpha)^2$ (which is polynomial in $\mathcal{S}(G, \alpha)$). □

Proposition 11.14. *Let $(G, \alpha) \in \mathcal{B}$ with G an elementary abelian p-group, $T : \mathcal{P}(G) \to \{\text{true}, \text{false}\}$ an efficient linear dependence tester, S an independent generating set for G, and $g \in G$.*

Finding the factorization of g with respect to S (i.e. finding the multiplicities of the elements in S such that the sum gives g) is possible using at most $\log_p |G| \cdot p$ invocations of T.

Proof. The multiplicities m_i (such that $\sum_{i=1}^{|S|} m_i S[i] = g$) are computed by the following algorithm:

1. Let $S' \leftarrow S$ and $r \leftarrow g$.
2. For $1 \leq i \leq |S|$:
   a) Let $s \leftarrow S[i]$ and $S' \leftarrow S' \setminus \{s\}$.
   b) For $0 \leq j < p$:
      i. Let $h \leftarrow r - j \cdot s$.
      ii. If $h \in S'$ or $T(S' \cup \{h\})$: set $m_i \leftarrow j$, $r \leftarrow h$ and `break`.

The outer loop runs $|S|$ times and the inner loop at most p times, thus the linear dependence test T is invoked at most $|S| \cdot p = \log_p |G| \cdot p$ times. □

Proposition 11.15. *Let $(G, \alpha) \in \mathcal{B}$ with G an elementary abelian p-group of order p^s and $\alpha = (A_1, \ldots, A_s) \in \Lambda(G)$ canonical of type $t(\alpha) = (p, p, \ldots, p)$.*

For all $(i_1, i_2, \ldots, i_s) \in \{2, 3, \ldots, p\}^s$, $S := \{A_j[i_j] \mid 1 \leq j \leq s\}$ is an independent generating set for G.

Proof. By Rédei's theorem (Theorem 8.1), at least one block of α is a subgroup. W.l.o.g. let $A_1 \leq G$. Then $\alpha' := (A_2, \ldots, A_s)$ is a logarithmic signature for G/A_1 (when interpreting all elements in α' as elements in G/A_1). For every element $a \in A_1 \setminus \{(0, \ldots, 0)\}$, we have $\langle a \rangle = A_1$. Pick an arbitrary element from $A_1 \setminus \{(0, \ldots, 0)\}$ and let it be the first element for the generating set S for G.

In α' there again exists a subgroup block (in G/A_1) by Rédei's theorem. W.l.o.g. let this be the block A_2. Then like above $\alpha'' := (A_3, \ldots, A_s) \in \Lambda(G/(A_1 + A_2))$. Let $a \in A_2 \setminus \{(0, \ldots, 0)\}$. Every element a' in A_2 (interpreted as element in G) can be written as $a' = h + k \cdot a$ with $h \in A_1$ and $k \in \mathbb{N}$. Consequently $A_1 + \langle a \rangle = A_1 + A_2$, so let a be the second element for the generating set S for G. We have $A_1 \cap \langle a \rangle = \{(0, \ldots, 0)\}$ (because $\mathrm{ord}(a) = p$ and $a \notin A_1$), so the first two generator elements are independent.

By iterating this idea, we see that no matter which element except $(0, \ldots, 0)$ we select in each block of α, the resulting set is always an independent generating set for G. □

A slightly generalized result can be found in Section 11.4.

11.4. Abelian Groups

Proposition 11.16. *Let $(G, \alpha) \in \mathcal{B}$ with G an abelian p-group, T an efficient subgroup membership test for subgroups defined by independent generators, $\emptyset \neq S$ a set of independent group elements (with $(0, \ldots, 0) \notin S$), and $g \in G$.*

Then computing the order of $g + \langle S \rangle$ in the factor group $G/\langle S \rangle$ is possible efficiently.

Proof. The order $m = \mathrm{ord}(g + \langle S \rangle)$ of $g + \langle S \rangle$ in $G/\langle S \rangle$ is computed by the following algorithm:

1. Set $m \leftarrow 1$ and $h \leftarrow g$.
2. Repeat:
   a) Use T to test whether $h \in \langle S \rangle$. If $h \in \langle S \rangle$: `break`.
   b) Set $h \leftarrow p \cdot h$ and $m \leftarrow m \cdot p$.

The loop is executed at most $\log_p |G|$ times. □

Proposition 11.17. *Let $(G, \alpha) \in \mathcal{B}$ with G abelian. Then we can efficiently find sets of generators for all p-Sylow subgroups of G.*

Proof. Let $|G| = p_1^{e_1} \cdots p_s^{e_s}$ with $p_i \in \mathbb{P}$ pairwise different and $e_i \in \mathbb{N}$. Define i_p to be the index of the prime p in the factorization of $|G|$, and $m_p := \prod_{i \in \{1,\ldots,s\} \setminus \{i_p\}} p_i^{e_i}$. Then $M_p := \{m_p \cdot g \mid g \in E(\alpha)\}$ is a generating set for the p-Sylow subgroup of G.

This can be seen as follows. It is clear that all elements in M_p have a p-power order, because each element from $E(\alpha)$ was multiplied by all other maximal prime powers (thus these components vanish); so, $\langle M_p \rangle$ is a p-subgroup. $\langle M_p \rangle$ is the full p-Sylow subgroup, because the other primes are all coprime to p (thus the p-components do not vanish). □

Proposition 11.18. *Let $(G, \alpha) \in \mathcal{B}$, where G is abelian of type $t(G) = (p_1, p_2, \ldots, p_s)$ (with $p_i \in \mathbb{P}$ for all $1 \leq i \leq s$, not necessarily pairwise different) and $\alpha = (A_1, \ldots, A_s)$ canonical of type $t(\alpha) = (p_1, p_2, \ldots, p_s)$. Define $M := \{p_1, \ldots, p_s\}$ and $m := \prod_{p \in M} p$ (such that $\gcd(m, p_i^k) = p_i$ for all $1 \leq i \leq s$ and $k \in \mathbb{N}$).*

For all $(i_1, i_2, \ldots, i_s)$ with $2 \leq i_1 \leq p_1, 2 \leq i_2 \leq p_2, \ldots, 2 \leq i_s \leq p_s$, the set $S := \{\frac{m}{|A_j|} \cdot A_j[i_j] \mid 1 \leq j \leq s\}$ is an independent generating set for G.

Proof. See the proof of Proposition 11.15. With respect to the factor group $G/(A_1 + \ldots + A_{j-1})$, the multiplication by $\frac{m}{|A_j|}$ just permutes the elements of the subgroup block A_j (because A_j is cyclic in $G/(A_1 + \ldots + A_{j-1})$ and $\gcd(\frac{m}{|A_j|}, |A_j|) = 1$). With respect to G, it ensures that the orders of all elements in S are prime numbers; this results in the elements in S being independent. □

11.5. Dihedral Groups

Theorem 11.19. *Let $(G, \alpha) \in \mathcal{B}$ with G a dihedral group of order $|G| \geq 6$. Define $n := \frac{|G|}{2}$.*

Finding some $\sigma \in G$ and $\tau \in G$ with $\operatorname{ord}(\sigma) = n$, $\operatorname{ord}(\tau) = 2$ and $\sigma\tau = \tau\sigma^{-1}$ is possible efficiently. Furthermore, for every $g \in G$ we can efficiently find $k, b \in \mathbb{N}_0$ such that $g = \sigma^k\tau^b$.

Proof. First of all, let $X := E(\alpha)$ and compute the set $X' := (X \cup \{\mathrm{id}\})^2 = \{xx' \mid x, x' \in (X \cup \{\mathrm{id}\})\}$. Let r be an arbitrary element in X' with $\operatorname{ord}(r) \geq 3$ (such an element must exist, because the *cyclic* normal subgroup of rotations has size $n \geq 3$, and X' contains all "partial" rotations). Clearly, r is a rotation and not the identity.

Now we partition $X' = R \dot\cup T$ such that R contains rotations only and T reflections only. In order to decide whether a $g \in X'$ is a rotation or a reflection, compute $g' := r^{-1}gr$; if $g' = g$, then g is a rotation, otherwise it is a reflection.

We have $\langle R \rangle \cong \mathbb{Z}_n$. Use Theorem 11.10 to construct a logarithmic signature $\beta \in \Lambda(\langle R \rangle)$ from R. Then use Theorem 11.7 (for $(\langle R \rangle, \beta)$) to find a $\sigma \in G$ with $\langle \sigma \rangle = \langle R \rangle$.

Pick an arbitrary element from T and call it τ. It does not matter which element we choose, because $\tau \mapsto \sigma^l\tau$ is an automorphism for all $l \in \mathbb{Z}_n$ by Lemma 2.1 (automorphism $\varphi_{1,l}$).

Having found elements σ, τ, we can now map arbitrary black box group elements to this representation. Let $g \in G$. We are looking for a $k \in \mathbb{N}_0$ and a $b \in \{0, 1\}$ such that $g = \sigma^k\tau^b$. First, test whether g is a rotation (g is a rotation if and only if $\sigma^{-1}g\sigma = g$). If g is a rotation, find a k with $g = \sigma^k$ using Theorem 11.8 and set b to 0. If g is a reflection, compute $g' := g \cdot \tau$, find a k with $g' = \sigma^k$ using Theorem 11.8 and set b to 1. □

Factoring. Let $(G, \alpha) \in \mathcal{B}$ with G a dihedral group, and $g \in G$. Factoring g with respect to α is as hard as in Chapter 9, because using Theorem 11.19 we can first map g and all group elements of α to the representation used in Chapter 9 and then apply the algorithms for dihedral groups.

11.6. Groups G of Order $|G| = 2^{n+1}$ with a Cyclic $N \trianglelefteq G$ of Order $|N| = 2^n$

Theorem 11.20. *Let $n \geq 3$ and $(G, \alpha) \in \mathcal{B}$ with G a non-abelian group of order $|G| = 2^{n+1}$ having a cyclic normal subgroup $N \trianglelefteq G$ of order $|N| = 2^n$ (we require that such an*

N exists, but do not require it to be given for example by generators), and α canonical of type $(2, 2, \ldots, 2)$.

Then α is tame.

Proof. By Section 2.3.5, a non-abelian group of order $|G| = 2^{n+1}$ having a cyclic normal subgroup $N \trianglelefteq G$ of order $|N| = 2^n$ is isomorphic to either $D_{2\cdot 2^n}$, $Q_{4\cdot 2^{n-1}}$, $\mathfrak{G}_{2\cdot 2^n}$ or $\mathfrak{D}_{2\cdot 2^n}$.

Our goal is to identify the group and map all elements to the representation in Section 2.3.5. α is then tame by Lemma 10.5.

First of all, observe the following for all $k \in \mathbb{Z}_{2^n}$. For $\sigma^k\tau \in D_{2\cdot 2^n}$, we have $\mathrm{ord}(\sigma^k\tau) = 2$, because $(\sigma^k\tau)^2 = \sigma^{k-k}\tau\tau = \mathrm{id}$. For $\sigma^k\tau \in Q_{4\cdot 2^{n-1}}$, we have $\mathrm{ord}(\sigma^k\tau) = 4$, because $(\sigma^k\tau)^4 = (\sigma^{k-k}\tau\tau)^2 = (\sigma^{2^{n-1}})^2 = \mathrm{id}$ (and $\sigma^{2^{n-1}} \neq \mathrm{id}$). For $\sigma^k\tau \in \mathfrak{D}_{2\cdot 2^n}$, we have $\mathrm{ord}(\sigma^k\tau) \in \{2, 4\}$, because $(\sigma^k\tau)^2 = \sigma^{k-k+k\cdot 2^{n-1}}\tau\tau = \sigma^{k\cdot 2^{n-1}}$ and this element either has order 1 (if $k \equiv 0 \pmod 2$) or order 2 (if $k \equiv 1 \pmod 2$). Thus, for these three groups $\mathrm{ord}(g) = 2^n$ for a $g \in G$ implies that $g \in N$.

Let $X := E(\alpha)$ and compute the set $X' := (X \cup \{\mathrm{id}\})^2 = \{xx' \mid x, x' \in (X \cup \{\mathrm{id}\})\}$. Let σ be an arbitrary element in X' with $\mathrm{ord}(\sigma) = 2^n$ (such an element must exist). By the above observation, for $D_{2\cdot 2^n}$, $Q_{4\cdot 2^{n-1}}$ and $\mathfrak{D}_{2\cdot 2^n}$, we get $\langle\sigma\rangle = N$. If $G = \mathfrak{G}_{2\cdot 2^n}$, it also is $\langle\sigma\rangle \cong N$ by Lemma 2.5.

Find an arbitrary $t \in X'$ with $\sigma^{-1}t\sigma \neq t$; then $t \notin \langle\sigma\rangle$. In order to see why this is correct, observe the following for all $k \in \mathbb{Z}_{2^n}$. If $G \cong D_{2\cdot 2^n}$ or $G \cong Q_{4\cdot 2^{n-1}}$, then $\sigma^{-1}\sigma^k\tau\sigma = \sigma^{k-2}\tau \neq \sigma^k\tau$. If $G \cong \mathfrak{D}_{2\cdot 2^n}$, then $\sigma^{-1}\sigma^k\tau\sigma = \sigma^{k-2+2^{n-1}}\tau \neq \sigma^k\tau$, because $-2 + 2^{n-1} \not\equiv 0 \pmod{2^n}$. If $G \cong \mathfrak{G}_{2\cdot 2^n}$, then $\sigma^{-1}\sigma^k\tau\sigma = \sigma^{k+2^{n-1}}\tau \neq \sigma^k\tau$.

- If $\mathrm{ord}(t) = 2$ and $\mathrm{ord}(\sigma t) = 2$, then $G \cong D_{2\cdot 2^n}$. Define $\tau := t$.

  This results in a representation of G compatible with α, because $\tau \mapsto \sigma^l\tau$ is an automorphism for all $l \in \mathbb{Z}_{2^n}$ by Lemma 2.4.

- If $\mathrm{ord}(t) = 4$ and $\mathrm{ord}(\sigma t) = 4$, then $G \cong Q_{4\cdot 2^{n-1}}$. Define $\tau := t$.

  This results in a representation of G compatible with α, because $\tau \mapsto \sigma^l\tau$ is an automorphism for all $l \in \mathbb{Z}_{2^n}$ by Lemma 2.4.

- If either $\mathrm{ord}(t) = 2 \wedge \mathrm{ord}(\sigma t) = 4$ or $\mathrm{ord}(t) = 4 \wedge \mathrm{ord}(\sigma t) = 2$, then $G \cong \mathfrak{D}_{2\cdot 2^n}$. If $\mathrm{ord}(t) = 2$ define $\tau := t$, otherwise define $\tau := \sigma t$.

  This results in a representation of G compatible with α. In order to see this, let τ' be the original imaginary element without a factor from N. $\tau' \mapsto \sigma^l\tau'$ is an automorphism for all even $l \in \mathbb{Z}_{2^n}$ by Lemma 2.4. $\mathrm{ord}(\tau) = 2$ implies that τ can be expressed as $\tau = \sigma^l\tau'$ with an even l, i.e. τ' can be mapped to τ using an automorphism.

- Otherwise we have $G \cong \mathfrak{G}_{2\cdot 2^n}$ (and either $\mathrm{ord}(t) = 2^n$ or $\mathrm{ord}(\sigma t) = 2^n$). The following procedure computes a τ that results in a representation of G compatible with α:

  1. Set $t' \leftarrow t$.
  2. Repeat:

a) Set $u \leftarrow \operatorname{ord}(t')$.

b) If $u = 2$: `break`.

c) Set $t' \leftarrow \sigma^{\frac{2^n}{u}} t'$.

3. Define $\tau := t'$.

When τ' was the imaginary element in the original representation, clearly τ is either τ' or $\sigma^{2^{n-1}}\tau'$. These two are equivalent, because $\tau' \mapsto \sigma^{2^{n-1}}\tau'$ obviously is an automorphism (we have $(\sigma^{2^{n-1}}\tau')^2 = \mathrm{id}$ and $\sigma^{2^{n-1}}\tau' = \tau'\sigma^{2^{n-1}}$).

Mapping elements to powers of σ and τ works exactly as in the proof of Theorem 11.19. □

12. Implementation / Program Documentation

Our program LogSig implements various algorithms related to logarithmic signatures.

LogSig is written in C#, using the .NET class library. It runs on all Windows systems with the Microsoft .NET Framework 4.0 (i.e. Windows ≥ XP; .NET 4.0 is required, because LogSig for instance uses the `BigInteger` class, which is only present in version 4.0 and higher) and all systems supported by Mono (e.g. Linux and Mac OS X).

Building. To build an executable application from the source code, do the following:

- Under Windows: open the `LogSig.sln` file using Microsoft Visual Studio or SharpDevelop[1] IDE, and build the solution ("Build" → "Rebuild Solution").
- Under Linux / Mac OS X: open the `LogSig.sln` file using MonoDevelop, and build the solution ("Build" → "Rebuild Solution").

12.1. Command Line Options

LogSig is a console application. The task to perform is specified using command line options/parameters.

Command line options/parameters must start with "`-`" or "`--`"; the two prefixes are equivalent. For example, the following mean exactly the same: `-t` and `--t`.

Some parameters support values. In order to separate the parameter name from its value, use "`:`" or "`=`". For example, `-r:200` and `-r=200` are equivalent.

Option/parameter names are case-insensitive (e.g. "`-t`" and "`-T`" are equivalent). Parameter values may be case-sensitive, depending on the parameter (e.g. the value of a password parameter "`-pw:`" is case-sensitive, whereas for example the value of a group parameter "`-g:`" is case-insensitive).

General command line options:

[1] http://www.icsharpcode.net/OpenSource/SD/Default.aspx

- "`-t`": Performs a self-test. In the self-test, various parts of the application are tested, including but not limited to numerical methods (factorization, string conversion, etc.) and cryptographic algorithms (e.g. the cryptographically secure pseudo-random number generator).
- "`-fct`": Enables factorization of numbers. Group and element orders are factorized before writing them to file/screen; e.g. instead of writing `4698`, LogSig writes `2*3^4*29`. Computing number factorizations can require a long time and can considerably slow down LogSig, so this should only be enabled for testing and analysis purposes.
- "`-mst1`": Prerequisite option for all commands related to generalized MST_1 algorithms (details in Section 12.2). Supported suboptions are:
  - "`-c`": Create a public/private key pair.
  - "`-e`": Encrypt a file.
  - "`-d`": Decrypt a file.

  These options are explained in detail in Section 12.2.1.

12.2. Implementation of Generalized MST_1

LogSig implements a public-key system based on the generalized version of MST_1 (Section 4.3). Files can be encrypted using a public key and decrypted using a private key.

Generalized MST_1 with LS-Gen. A public key consists of a logarithmic signature generated by LS-Gen (Section 6.5). The corresponding private key consists of the same logarithmic signature and additionally some information how this logarithmic signature was created.

LS-Gen requires random numbers during the generation of a logarithmic signature. These are generated by a cryptographically secure pseudo-random number generator (CSPRNG) based on the Advanced Encryption Standard (AES) block cipher in counter (CTR) mode.

The seed for this generator (an AES key and an initial counter value) is basically the private key. Starting with a tame logarithmic signature and knowing the seed, LS-Gen can construct the public logarithmic signature again (i.e. the seed represents the information how the public logarithmic signature was generated) and allows to compute factorizations of arbitrary group elements.

Encryption/Decryption. When using a high rounds count for LS-Gen, the decryption is rather slow. Thus, files are encrypted/decrypted using a symmetric block cipher, namely AES, and the key for it is derived from the factorization of a random group element with respect to the public logarithmic signature.

In detail, LogSig does the following in order to encrypt a file:

- Load the public key file containing a logarithmic signature for a group G.
- Generate a random initialization vector (IV) and a random number $0 \leq s < |G|$.
- Map s to a vector of block indices in the logarithmic signature, using a mixed radix conversion where the bases are the element counts in the blocks. Multiply the logarithmic signature elements at these indices to obtain a group element $g \in G$.
- Write the header of the encrypted file. The header contains a file type identifier, a version number, the element g and the IV.
- Convert s to a byte array and hash this byte array using SHA-256. The resulting 256-bit hash is used as key for AES. Symmetrically encrypt the file now. LogSig uses the CBC block cipher mode of operation and a PKCS7 padding.

Note that s (which effectively is the factorization of g in the public logarithmic signature) is not stored in the encrypted file; s must be erased securely after the encrypted file has been created.

In order to decrypt a file, LogSig does the following:

- Load the private key file and the header of the encrypted file, i.e. especially the element $g \in G$ and the IV.
- Run LS-Gen to obtain the factorization for g. Undo the mixed radix representation to reconstruct $0 \leq s < |G|$.
- Convert s to a byte array and hash this byte array using SHA-256. Decrypt the file symmetrically now, using the 256-bit hash as key and the loaded IV.

12.2.1. Command Line Examples

Generating a key pair. In order to create a public/private key pair, invoke LogSig with the options "`-mst1`" and "`-c`".

By default, the names of the output files are *Generated.pub* and *Generated.prv*. To change "Generated", you can specify a different base name using the command line parameter "`-base:`".

The group must be specified using the command line parameter "`-g:`". The following are valid, built-in group names (plugins can provide additional groups):

- `Dih` or `D`: dihedral group.
- `Sym` or `S`: symmetric group.
- `Alt` or `A`: alternating group.
- `C` or `Z`: cyclic group.
- `Quat` or `Q`: generalized quaternion group.

Additionally, direct products/sums and wreath products are supported. To create a direct product/sum, separate the groups by an "`x`". For specifying the wreath product of two groups, separate them by a "`w`" (the second group is the permutation group acting on itself).

The size of the group must be appended to its name. Usually, the group order (i.e. the number of elements) is appended. However, for the symmetric and the alternating group, the degree (size of the set that the permutation group acts on) needs to be specified.

For specifying the size, "`^`" and "`*`" can be used; e.g. you can write `2^32*3` instead of 12884901888). If you are using the factorization notation, enclose the number in quotes (`"`).

Examples:

- "`-g:Sym4`" – The symmetric group Sym(4).
- "`-g:Z2xZ3xZ3xSym5`" – The direct product $\mathbb{Z}_2 \times \mathbb{Z}_3 \times \mathbb{Z}_3 \times \mathrm{Sym}(5)$.
- "`-g:"D2*32"`" – The dihedral group $D_{2\cdot 32}$.
- "`-g:"D2*2^8xZ2xZ3"`" – The direct product $D_{2\cdot 2^8} \times \mathbb{Z}_2 \times \mathbb{Z}_3$.
- "`-g:D64wZ11`" – The wreath product $D_{2\cdot 32} \wr \mathbb{Z}_{11}$.

The default number of LS-Gen rounds is 5000. This can be changed using the command line parameter "`-r:`".

The generated private key file is encrypted using a master password for security reasons. This master password must be specified using the command line parameter "`-pw:`". If the password contains a space or any special command line character, it must be enclosed in quotes.

Example:

```
LogSig.exe -mst1 -c -g:"D2*2^128" -r:200 -pw:abc
```

Generates a key pair for a logarithmic signature of $D_{2\cdot 2^{128}}$. For the generation of the logarithmic signature, 200 LS-Gen rounds are used. The private key file is protected using the master password "abc".

Encrypting a file. To encrypt one or more files, invoke LogSig with the options "`-mst1`" and "`-e`". The files to encrypt are passed as arguments (without a parameter name).

The public key file path has to be specified using the parameter "`-key:`".

By default, the name of the output file is the name of the input file name plus ".enc". However, you can freely specify a different output file path using the parameter "`-outfile:`". This parameter of course only makes sense when encrypting exactly one file; if more files are encrypted, this parameter cannot be used.

Example:

`LogSig.exe -mst1 -e -key:Generated.pub TestFile.txt`
Encrypts the file *TestFile.txt* using the public key stored in *Generated.pub*. The encrypted data is saved to the file *TestFile.txt.enc*; the input file is not modified.

Decrypting a file. For decrypting, invoke LogSig with the options "`-mst1`" and "`-d`". Similar to the encryption call, the files to decrypt are passed as arguments (without a parameter name) and the private key file has to be passed using "`-key:`". The decrypted file is written to a file named like the input file plus ".dec". Like before, this can be overridden using "`-outfile:`".

The master password required for decrypting the private key has to be passed using the parameter "`-pw:`".

Example:

`LogSig.exe -mst1 -d -key:Generated.prv -pw:abc TestFile.txt.enc`
Decrypts the private key file *Generated.prv* using the master password "abc" and then uses this information to decrypt the file *TestFile.txt.enc*. The output is written to *TestFile.txt.enc.dec*.

12.2.2. Sample Key Files

A public key file is an XML file containing definitions for the group being used (type and order) and a logarithmic signature. A private key XML file additionally contains the seed, round count and transformation input/output relations for LS-Gen.

For example, the following public key XML file defines a logarithmic signature for the dihedral group of order $270 = 2 \cdot 3^3 \cdot 5$ (such a small group is used just for giving an example for the file format; in a real system a much larger group and a larger round count would of course be used):

```
<?xml version="1.0" encoding="utf-8"?>
<PkcPublicKey xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <Group>D2*135</Group>
    <LogSig>((id, 132, 36t), (id, 117, 6t, 105t, 36), (id, 42t),
        (id, 85t, 46), (id, 90, 40t))</LogSig>
</PkcPublicKey>
```

The corresponding private key XML file could look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<PkcPrivateKey xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
    <Group>D2*135</Group>
    <LogSig>((id, 132, 36t), (id, 117, 6t, 105t, 36), (id, 42t),
        (id, 85t, 46), (id, 90, 40t))</LogSig>
    <GeneratorKey>I/WzFoY7Ro6fianfTiX89+vAo+RDBiHZyDoRe4+H5gc=
        </GeneratorKey>
    <GeneratorCtr>wtvMtu7M98PhbHS4wrxsKA==</GeneratorCtr>
    <GeneratorRounds>200</GeneratorRounds>
    <GeneratorTrfIORels>(T:130t/52)</GeneratorTrfIORels>
</PkcPrivateKey>
```

Note that `GeneratorKey` and `GeneratorCtr` are encrypted using the master password.

12.3. Factoring in Abelian Groups

The factorization algorithm in Section 8.4/8.5 (for logarithmic signatures with periodic blocks on each recursion level) and the generic one in Section 8.7 (for arbitrary block sequences of abelian groups) have been implemented.

- In order to use the algorithm of Section 8.4/8.5, pass the command line option "`-AbFac`".
- In order to use the algorithm of Section 8.7, pass the command line option "`-AbFacEx`".

The group can be specified using the parameter "`-g:`" (must be specified as a direct sum of cyclic groups), the logarithmic signature or block sequence in the parameter "`-ls:`" (for "`-AbFac`" it must be a logarithmic signature with periodic blocks on each recursion level; for "`-AbFacEx`" it may be any arbitrary block sequence), and the element to factor in the parameter "`-el:`".

Example 12.1. Let $G = \mathbb{Z}_8 \oplus \mathbb{Z}_4$, $g = (5,3) \in G$, and

$$\alpha = \binom{(0,0)}{(2,1)}\binom{(0,0)}{(4,0)}\binom{(0,0)}{(1,2)}\binom{(0,0)}{(2,0)}\binom{(0,0)}{(4,2)} \in \Lambda(G).$$

In order to compute the factorization of g with respect to α, invoke LogSig as follows:

```
LogSig.exe -AbFac -g:Z8xZ4 -ls:"(((0,0), (2,1)), ((0,0), (4,0)),
((0,0), (1,2)), ((0,0), (2,0)), ((0,0), (4,2)))" -el:(5,3)
```

The program outputs:

```
(5,3) = (2,1) + (0,0) + (1,2) + (2,0) + (0,0).
```

12.4. Plugin Architecture

LogSig was written with extensibility in mind; it has a plugin architecture. Plugins can add more application modules (handlers for command line arguments), as well as new groups.

An example of a complete plugin adding support for $\mathbb{Z}_n$ groups (once more) can be found in Appendix B.

13. Further Research

As we have shown in Section 8.9, amalgamated transversal logarithmic signatures and various constructions for aperiodic logarithmic signatures are tame. Furthermore, for abelian groups we have pointed out that our generic factorization algorithm in Section 8.7 usually works efficiently for logarithmic signatures generated by LS-Gen.

Thus it remains an open problem whether it is possible to efficiently construct possibly wild logarithmic signatures for abelian groups (and whether the construction allows to be used as a trap-door).

LS-Gen can generate logarithmic signatures for non-abelian groups. Although the algorithm uses a few powerful transformations during the construction process, in practice (when implementing a cryptosystem based on wild logarithmic signatures) it is a good idea to implement additional group-specific transformations for efficiency (for an example, see the note on the efficiency of block substitutions in Section 5.1.8). An example of a group-specific transformation for dihedral groups is the reflection spreading in Algorithm 9.9 (based on our Theorem 9.2 about size-permutability).

In the future, more group-specific transformations may be presented.

The development of factorization algorithms for more groups and logarithmic signature types will be the subject of further research. Especially interesting might be direct products (e.g. the group D_{2n}^2).

A. Factorization Algorithm for Abelian Groups

The following is the source code of the generic factorization Algorithm 8.30 for abelian groups described in Section 8.7.

Listing A.1: DmMultiZnGroup.FacEx.cs

```
1  // LogSig - Logarithmic Signature Utility
2  // Copyright (C) 2011-2015 Dominik Reichl <info@dominik-reichl.de>
3
4  using System;
5  using System.Collections.Generic;
6  using System.Linq;
7  using System.Text;
8  using System.Numerics;
9  using System.Diagnostics;
10
11 namespace LogSig.DM.MultiZnGroup
12 {
13     public sealed partial class DmMultiZnGroup : IDmGroup<DmMultiZnGroupElement>
14     {
15         /// <summary>
16         /// Compute all factorizations of <paramref name="el" /> with
17         /// respect to <paramref name="bs" />.
18         /// </summary>
19         /// <param name="bs">An arbitrary block sequence of elements
20         /// in <paramref name="grp" /> (not necessarily a logarithmic signature
21         /// or multiple factorization of <paramref name="grp" />).</param>
22         /// <param name="grp">Group.</param>
23         /// <param name="el">Element to factor.</param>
24         /// <returns>All factorizations of <paramref name="el" />.</returns>
25         public static List<int[]> FactorizeAbEx(
26             DmBlockSeq<DmMultiZnGroupElement> bs, DmMultiZnGroup grp,
27             DmMultiZnGroupElement el, DmTraceInfo ti)
28         {
29             return FactorizeAbEx(bs, grp, el,
30                 new DmBlockSeq<DmMultiZnGroupElement>(), ti);
31         }
32
33         private static List<int[]> FactorizeAbEx(
34             DmBlockSeq<DmMultiZnGroupElement> bs, DmMultiZnGroup grp,
35             DmMultiZnGroupElement el, DmBlockSeq<DmMultiZnGroupElement> lsDivChain,
36             DmTraceInfo ti)
37         {
38             List<int[]> lRes = new List<int[]>();
39             NormalChainEq eq = new NormalChainEq(grp, lsDivChain);
40
41             if(bs.Blocks.Count == 1)
42             {
43                 for(int i = 0; i < bs.Blocks[0].Elements.Count; ++i)
44                 {
```

```
45                      if(eq.Equals(bs.Blocks[0].Elements[i], el))
46                          lRes.Add(new int[] { i });
47                  }
48                  return UpdTrace(lRes, ti);
49              }
50
51              for(int x = 0; x < bs.Blocks.Count; ++x)
52              {
53                  DmLogSigBlock<DmMultiZnGroupElement> blk = bs.Blocks[x];
54                  List<DmMultiZnGroupElement> lUni = blk.Elements.Distinct(
55                      eq).ToList();
56                  if(blk.Elements.Count > lUni.Count)
57                  {
58                      DmBlockSeq<DmMultiZnGroupElement> bsNew = bs.Clone();
59                      bsNew.Blocks[x].Elements.Clear();
60                      bsNew.Blocks[x].Elements.AddRange(lUni);
61
62                      List<int[]> lSubFac = FactorizeAbEx(bsNew, grp, el,
63                          lsDivChain, ti);
64                      return UpdTrace(EnumAndMatch(bs, grp, el, eq, lSubFac,
65                          x, false), ti);
66                  }
67              }
68
69              for(int x = 0; x < bs.Blocks.Count; ++x)
70              {
71                  DmLogSigBlock<DmMultiZnGroupElement> blk = bs.Blocks[x];
72                  if(blk.Elements.Count == 1)
73                  {
74                      DmBlockSeq<DmMultiZnGroupElement> bsNew = bs.Clone();
75                      bsNew.Blocks.RemoveAt(x);
76
77                      List<int[]> lSubFac = FactorizeAbEx(bsNew, grp, el,
78                          lsDivChain, ti);
79                      return UpdTrace(EnumAndMatch(bs, grp, el, eq, lSubFac,
80                          x, true), ti);
81                  }
82              }
83
84              List<DmMultiZnGroupElement> lN = ChooseNormalSubgroup(bs, grp, eq);
85
86              DmBlockSeq<DmMultiZnGroupElement> lsDivChainN = lsDivChain.Clone();
87              lsDivChainN.AddBlock(lN.ToArray());
88
89              List<int[]> lSubFacN = FactorizeAbEx(bs, grp, el, lsDivChainN, ti);
90              foreach(int[] vCand in lSubFacN)
91              {
92                  if(eq.Equals(bs.MultiplyElements(new List<int>(vCand), grp), el))
93                      lRes.Add(vCand);
94              }
95
96              return UpdTrace(lRes, ti);
97          }
98
99          private static List<int[]> UniqueFacs(List<int[]> lFacs)
100         {
101             return lFacs.Distinct(new ArraysEqualComparer<int>()).ToList();
102         }
103
104         private static List<int[]> UpdTrace(List<int[]> l, DmTraceInfo ti)
105         {
106             if(l.Count > ti.MaxWidth) ti.MaxWidth = l.Count;
```

```
107             return l;
108         }
109
110         private static List<int[]> EnumAndMatch(
111             DmBlockSeq<DmMultiZnGroupElement> bs, DmMultiZnGroup grp,
112             DmMultiZnGroupElement el, NormalChainEq eq, List<int[]> lFacs,
113             int x, bool bInsert)
114         {
115             List<int[]> lRes = new List<int[]>();
116
117             foreach(int[] vCand in lFacs)
118             {
119                 for(int j = 0; j < bs.Blocks[x].Elements.Count; ++j)
120                 {
121                     List<int> lFac = new List<int>(vCand);
122
123                     if(bInsert) lFac.Insert(x, j);
124                     else lFac[x] = j;
125
126                     if(eq.Equals(bs.MultiplyElements(lFac, grp), el))
127                         lRes.Add(lFac.ToArray());
128                 }
129             }
130
131             return UniqueFacs(lRes);
132         }
133
134         private sealed class ArraysEqualComparer<T> : IEqualityComparer<T[]>
135             where T : IEquatable<T>
136         {
137             public bool Equals(T[] x, T[] y)
138             {
139                 if((x == null) || (y == null))
140                 {
141                     Debug.Assert(false);
142                     return false;
143                 }
144
145                 if(x.Length != y.Length) return false;
146
147                 for(int i = 0; i < x.Length; ++i)
148                 {
149                     if(!x[i].Equals(y[i])) return false;
150                 }
151
152                 return true;
153             }
154
155             public int GetHashCode(T[] obj)
156             {
157                 if(obj == null) { Debug.Assert(false); return 0; }
158
159                 return obj.Sum(o => o.GetHashCode());
160             }
161         }
162
163         private sealed class ElemWithRatings
164         {
165             public DmMultiZnGroupElement Element;
166             public List<BigInteger> Ratings;
167
168             public ElemWithRatings(DmMultiZnGroupElement e,
```

```
169                 List<BigInteger> lRatings)
170             {
171                 this.Element = e;
172                 this.Ratings = lRatings;
173             }
174         }
175 
176         private static List<DmMultiZnGroupElement> ChooseNormalSubgroup(
177             DmBlockSeq<DmMultiZnGroupElement> bs, DmMultiZnGroup grp,
178             NormalChainEq eq)
179         {
180             IEnumerable<DmMultiZnGroupElement> eC = GetLowOrderElements(bs, grp,
181                 eq);
182 
183             List<ElemWithRatings> lR = new List<ElemWithRatings>();
184             foreach(DmMultiZnGroupElement e in eC)
185             {
186                 NormalChainEq eqSub = AppendSubgroup(grp, eq, e);
187                 List<BigInteger> lRating = EstSeqComplexity(bs, grp, eqSub);
188                 if(lRating.TrueForAll(bi => bi.IsOne))
189                     return GetCyclicSpan(grp, eq, e).Elements;
190                 lRating[0] *= ElementOrder(e, grp, eq);
191                 lR.Add(new ElemWithRatings(e, lRating));
192             }
193 
194             BigInteger biEta = lR.Min(r => r.Ratings[0]);
195             lR = lR.Where(r => (r.Ratings[0] == biEta)).ToList();
196 
197             BigInteger biTheta = lR.Min(r => r.Ratings[1]);
198             lR = lR.Where(r => (r.Ratings[1] == biTheta)).ToList();
199 
200             if(lR.Count == 0) throw new Exception("lR.Count == 0");
201             int ri = Program.CryptoRandom.GenerateInt32(lR.Count);
202             return GetCyclicSpan(grp, eq, lR[ri].Element).Elements;
203         }
204 
205         private static IEnumerable<DmMultiZnGroupElement> GetLowOrderElements(
206             DmBlockSeq<DmMultiZnGroupElement> bs, DmMultiZnGroup grp,
207             NormalChainEq eq)
208         {
209             List<DmMultiZnGroupElement> lM = new List<DmMultiZnGroupElement>();
210             for(int x = 0; x < bs.Blocks.Count; ++x)
211             {
212                 var blk = bs.Blocks[x].Elements;
213                 for(int j = 0; j < (blk.Count - 1); ++j)
214                 {
215                     for(int k = j + 1; k < blk.Count; ++k)
216                     {
217                         var m = grp.Multiply(grp.Invert(blk[j]), blk[k]);
218                         if(lM.TrueForAll(el => !eq.Equals(el, m)))
219                             lM.Add(m);
220                     }
221                 }
222             }
223 
224             BigInteger q = new BigInteger(bs.Blocks.Max(
225                 blk => blk.Elements.Count));
226 
227             int d = 1;
228             List<DmMultiZnGroupElement> lC = new List<DmMultiZnGroupElement>();
229             while(lC.Count == 0)
230             {
```

```
231                 lC = lM.Where(e => ((BigInteger.Pow(q, d - 1) < ElementOrder(e,
232                     grp, eq)) && (ElementOrder(e, grp, eq) <=
233                     BigInteger.Pow(q, d)))).ToList();
234                 ++d;
235             }
236 
237             for(int x = lC.Count - 1; x > 0; --x)
238             {
239                 BigInteger ordX = ElementOrder(lC[x], grp, eq);
240                 var blkSpanX = GetCyclicSpan(grp, eq, lC[x]);
241 
242                 bool bValid = true;
243                 for(int y = 0; y < x; ++y)
244                 {
245                     if((ElementOrder(lC[y], grp, eq) == ordX) &&
246                         (blkSpanX.Elements.FindIndex(el => eq.Equals(el,
247                         lC[y])) >= 0))
248                     {
249                         bValid = false;
250                         break;
251                     }
252                 }
253 
254                 if(bValid) yield return lC[x];
255             }
256 
257             yield return lC[0];
258         }
259 
260         private static BigInteger ElementOrder(DmMultiZnGroupElement el,
261             DmMultiZnGroup grp, NormalChainEq eq)
262         {
263             if(eq.Equals(el, grp.Identity)) return BigInteger.One;
264 
265             List<BigInteger> lFactors = new List<BigInteger>(grp.OrderFactors);
266             int p = lFactors.Count - 1;
267 
268             while(p >= 0)
269             {
270                 BigInteger biPow = lFactors.Aggregate(BigInteger.One,
271                     (prod, f) => prod * f);
272                 biPow /= lFactors[p];
273                 if(eq.Equals(grp.Pow(el, biPow), grp.Identity))
274                     lFactors.RemoveAt(p);
275 
276                 --p;
277             }
278 
279             return lFactors.Aggregate(BigInteger.One, (prod, f) => prod * f);
280         }
281 
282         private static DmLogSigBlock<DmMultiZnGroupElement> GetCyclicSpan(
283             DmMultiZnGroup grp, NormalChainEq eq, DmMultiZnGroupElement g)
284         {
285             var blkSubgroup = new DmLogSigBlock<DmMultiZnGroupElement>();
286             DmMultiZnGroupElement s = grp.Identity;
287             do
288             {
289                 blkSubgroup.Elements.Add(s);
290                 s = grp.Multiply(s, g);
291             }
292             while(!eq.Equals(s, grp.Identity));
```

```
293             return blkSubgroup;
294         }
295 
296         private static NormalChainEq AppendSubgroup(DmMultiZnGroup grp,
297             NormalChainEq eq, DmMultiZnGroupElement g)
298         {
299             DmBlockSeq<DmMultiZnGroupElement> lsNew = eq.NormalChain.Clone();
300             lsNew.Blocks.Add(GetCyclicSpan(grp, eq, g));
301             return new NormalChainEq(grp, lsNew);
302         }
303 
304         private static List<BigInteger> EstSeqComplexity(
305             DmBlockSeq<DmMultiZnGroupElement> bs, DmMultiZnGroup grp,
306             NormalChainEq eq)
307         {
308             var bsSmp = SimplifySeq(bs, grp, eq);
309             if(bsSmp.Blocks.Count <= 1)
310                 return new List<BigInteger>(new BigInteger[] {
311                     BigInteger.One, BigInteger.One });
312 
313             BigInteger u = bsSmp.Blocks.Aggregate(BigInteger.One, (prod, blk) =>
314                 (prod * blk.Elements.Count));
315             BigInteger s = bsSmp.Blocks.Aggregate(BigInteger.Zero, (so, blk) =>
316                 (so + blk.Elements.Aggregate(BigInteger.Zero, (si, e) =>
317                 (si + ElementOrder(e, grp, eq)))));
318 
319             var eC = GetLowOrderElements(bsSmp, grp, eq);
320             foreach(DmMultiZnGroupElement g in eC)
321             {
322                 NormalChainEq eqSub = AppendSubgroup(grp, eq, g);
323                 var bsSub = SimplifySeq(bsSmp, grp, eqSub);
324                 if((ElementOrder(g, grp, eq) * bsSub.Blocks.Aggregate(
325                     BigInteger.One, (prod, blk) => (prod *
326                     blk.Elements.Count))) <= u)
327                     return EstSeqComplexity(bsSmp, grp, eqSub);
328             }
329 
330             List<BigInteger> lRes = new List<BigInteger>(2);
331             lRes.Add(u);
332             lRes.Add(s);
333             return lRes;
334         }
335 
336         private static DmBlockSeq<DmMultiZnGroupElement> SimplifySeq(
337             DmBlockSeq<DmMultiZnGroupElement> bs, DmMultiZnGroup grp,
338             NormalChainEq eq)
339         {
340             var bsNew = new DmBlockSeq<DmMultiZnGroupElement>();
341             for(int x = 0; x < bs.Blocks.Count; ++x)
342                 bsNew.AddBlock(bs.Blocks[x].Elements.Distinct(eq).ToArray());
343             for(int x = bs.Blocks.Count - 1; x >= 0; --x)
344             {
345                 if(bsNew.Blocks[x].Elements.Count == 1)
346                     bsNew.Blocks.RemoveAt(x);
347             }
348             return bsNew;
349         }
350 
351         private sealed class NormalChainEq :
352             IEqualityComparer<DmMultiZnGroupElement>
353         {
354             private readonly DmMultiZnGroup m_grp;
```

```
355             private readonly DmMultiZnGroupElement[] m_v;
356
357             private readonly DmBlockSeq<DmMultiZnGroupElement> m_ls;
358             public DmBlockSeq<DmMultiZnGroupElement> NormalChain
359             {
360                 get { return m_ls; }
361             }
362
363             public NormalChainEq(DmMultiZnGroup grp,
364                 DmBlockSeq<DmMultiZnGroupElement> lsNormalChain)
365             {
366                 if(grp == null) throw new ArgumentNullException("grp");
367                 m_grp = grp;
368
369                 m_v = new DmMultiZnGroupElement[grp.Mods.Length];
370                 m_ls = Simplify(grp, lsNormalChain);
371             }
372
373             private DmBlockSeq<DmMultiZnGroupElement> Simplify(DmMultiZnGroup grp,
374                 DmBlockSeq<DmMultiZnGroupElement> lsChain)
375             {
376                 if(lsChain == null) throw new ArgumentNullException("lsChain");
377
378                 DmBlockSeq<DmMultiZnGroupElement> ls = lsChain.Clone();
379
380                 for(int c = 0; c < grp.Mods.Length; ++c)
381                 {
382                     for(int x = 0; x < ls.Blocks.Count; ++x)
383                     {
384                         DmMultiZnGroupElement sMin = null;
385                         foreach(DmMultiZnGroupElement el in ls.Blocks[x].Elements)
386                         {
387                             if(el.Values[c].IsZero) continue;
388                             if(!el.Values.Take(c).All(bi => bi.IsZero)) continue;
389
390                             if((sMin == null) || (el.Values[c] < sMin.Values[c]))
391                                 sMin = el;
392                         }
393                         if(sMin == null) continue;
394
395                         m_v[c] = sMin;
396
397                         BigInteger sc = sMin.Values[c];
398                         Debug.Assert((sc >= 0) && (sc < grp.Mods[c]));
399                         for(int xr = x + 1; xr < ls.Blocks.Count; ++xr)
400                         {
401                             for(int yr = 0; yr < ls.Blocks[xr].Elements.Count;
402                                 ++yr)
403                             {
404                                 var h = ls.Blocks[xr].Elements[yr];
405                                 BigInteger hc = h.Values[c];
406                                 Debug.Assert((hc >= 0) && (hc < grp.Mods[c]));
407
408                                 if(sc <= hc)
409                                 {
410                                     var sMulti = grp.Pow(sMin, hc / sc);
411                                     Debug.Assert(!sMulti.Equals(grp.Identity));
412
413                                     var gRed = grp.Multiply(h, grp.Invert(sMulti));
414                                     ls.Blocks[xr].Elements[yr] = gRed;
415                                     Debug.Assert(gRed.Values[c] < sc);
416                                 }
```

```
417                             }
418                         }
419                     }
420                 }
421
422                 return ls;
423             }
424
425             public bool Equals(DmMultiZnGroupElement x, DmMultiZnGroupElement y)
426             {
427                 DmMultiZnGroupElement d = m_grp.Multiply(x, m_grp.Invert(y));
428
429                 for(int c = 0; c < m_grp.Mods.Length; ++c)
430                 {
431                     if(d.Values[c] == 0) continue;
432
433                     DmMultiZnGroupElement s = m_v[c];
434                     if(s == null) return false;
435
436                     BigInteger q = d.Values[c] / s.Values[c];
437
438                     DmMultiZnGroupElement sMul = m_grp.Pow(s, q);
439                     d = m_grp.Multiply(d, m_grp.Invert(sMul));
440
441                     if(!d.Values[c].IsZero) return false;
442                 }
443
444                 Debug.Assert(d.Values.All(bi => bi.IsZero));
445                 return true;
446             }
447
448             public int GetHashCode(DmMultiZnGroupElement obj)
449             {
450                 return 0; // Force that the Equals method is used
451             }
452         }
453     }
454 }
```

B. Sample Plugin: $\mathbb{Z}_n$ Group Provider

The following, complete source code of a plugin for LogSig (Chapter 12) adds support for $\mathbb{Z}_n$ groups.

LogSig does already support $\mathbb{Z}_n$ groups out of the box (by using `C` or `Z` as group name); this plugin just demonstrates how a plugin can look like.

When the source code is compiled and the DLL file placed in the directory of LogSig, all group parameters (e.g. "`-g:`") additionally support `Res` as group name, and all group-generic procedures (cryptosystem key pair generation, encryption, decryption, etc.) can use this group.

Listing B.1: LsZnGroupExt.cs

```
1  // LogSig - Logarithmic Signature Utility
2  // Copyright (C) 2011-2015 Dominik Reichl <info@dominik-reichl.de>
3
4  using System;
5  using System.Collections.Generic;
6  using System.Text;
7  using System.Numerics;
8
9  using LogSig.Crypto;
10 using LogSig.DM;
11 using LogSig.Plugins;
12
13 namespace LsZnGroup
14 {
15     public sealed class LsZnGroupExt : LsPlugin
16     {
17         public override bool Initialize(ILsPluginHost host)
18         {
19             if(host == null) return false;
20
21             host.GroupFactories.Add(new ZnGroupFactory());
22             return true;
23         }
24     }
25
26     public sealed class ZnGroupFactory : IDmGroupFactory
27     {
28         public object CreateInstance(string strGroup)
29         {
30             BigInteger? biOrder = NumUtil.GetPrefixedNumber(strGroup, "Res");
31             if(!biOrder.HasValue) return null;
32
33             return new ZnGroup(biOrder.Value);
34         }
35     }
36
37     public sealed class ZnGroupElement : IDmGroupElement<ZnGroupElement>
38     {
```

```
39          private int? m_hash = null;
40
41          private readonly BigInteger m_n;
42          public BigInteger Value { get { return m_n; } }
43
44          public ZnGroupElement(BigInteger biValue)
45          {
46              m_n = biValue;
47          }
48
49          public ZnGroupElement Clone()
50          {
51              return (ZnGroupElement)MemberwiseClone();
52          }
53
54          public override string ToString()
55          {
56              return NumUtil.FactorsToString(m_n, true);
57          }
58
59          public static ZnGroupElement Parse(string str)
60          {
61              return new ZnGroupElement(NumUtil.Parse(str));
62          }
63
64          public override int GetHashCode()
65          {
66              if(m_hash.HasValue) return m_hash.Value;
67
68              int h = m_n.GetHashCode();
69
70              m_hash = h;
71              return h;
72          }
73
74          public override bool Equals(object obj)
75          {
76              return Equals(obj as ZnGroupElement);
77          }
78
79          public bool Equals(ZnGroupElement other)
80          {
81              if(other == null) return false;
82              return (m_n == other.m_n);
83          }
84
85          public int CompareTo(ZnGroupElement other)
86          {
87              if(other == null) return 1;
88              return m_n.CompareTo(other.m_n);
89          }
90      }
91
92      public sealed class ZnGroup : IDmGroup<ZnGroupElement>
93      {
94          public string Name
95          {
96              get { return ("Res" + NumUtil.FactorsToString(m_biOrder, true)); }
97          }
98
99          private readonly BigInteger m_biOrder;
100         public BigInteger Order { get { return m_biOrder; } }
```

```
101
102         private ZnGroupElement m_id = new ZnGroupElement(0);
103         public ZnGroupElement Identity { get { return m_id; } }
104
105         public ZnGroup(BigInteger biOrder) { m_biOrder = biOrder; }
106
107         public ZnGroupElement ParseElement(string str)
108         {
109             return ZnGroupElement.Parse(str);
110         }
111
112         public ZnGroupElement Multiply(ZnGroupElement a, ZnGroupElement b)
113         {
114             return new ZnGroupElement((a.Value + b.Value) % m_biOrder);
115         }
116
117         public ZnGroupElement Invert(ZnGroupElement t)
118         {
119             if(t.Value.IsZero) return t;
120             return new ZnGroupElement(m_biOrder - t.Value);
121         }
122
123         public ZnGroupElement GenerateRandomElement(CryptoRandomGen rand)
124         {
125             return new ZnGroupElement(rand.GenerateBigInt(m_biOrder));
126         }
127
128         public ZnGroupElement NextElement(ZnGroupElement t)
129         {
130             return new ZnGroupElement((t.Value + BigInteger.One) % m_biOrder);
131         }
132
133         public BigInteger ElementToBigInt(ZnGroupElement t)
134         {
135             return t.Value;
136         }
137
138         public ZnGroupElement BigIntToElement(BigInteger biElem)
139         {
140             return new ZnGroupElement(biElem);
141         }
142
143         public DmBlockSeq<ZnGroupElement> CreateDefaultSig()
144         {
145             var ls = new DmBlockSeq<ZnGroupElement>();
146
147             BigInteger n = m_biOrder, p = 2, pp = 1;
148             while(n != 1)
149             {
150                 if((n % p).IsZero)
151                 {
152                     var blk = ls.AddBlock();
153                     for(int i = 0; i < p; ++i)
154                         blk.Elements.Add(new ZnGroupElement(pp * i));
155
156                     pp *= p;
157                     n /= p;
158                     continue;
159                 }
160
161                 ++p;
162             }
```

```
163
164             return ls;
165         }
166
167         public int[] FactorInDefaultSig(ZnGroupElement t)
168         {
169             DmBlockSeq<ZnGroupElement> ls = CreateDefaultSig();
170
171             int[] vFactorization = new int[ls.Blocks.Count];
172
173             BigInteger biRot = t.Value;
174             for(int i = 0; i < vFactorization.Length; ++i)
175             {
176                 int p = ls.Blocks[i].Elements.Count;
177                 vFactorization[i] = (int)(biRot % p);
178                 biRot /= p;
179             }
180
181             return vFactorization;
182         }
183     }
184 }
```

Bibliography

[Bab84] L. Babai, E. Szemerédi, *On the Complexity of Matrix Group Problems I*, in Proceedings of the 25th Annual Symposium on Foundations of Computer Science, pp. 229–240, 1984.
doi:10.1109/SFCS.1984.715919

[Bab95] L. Babai, G. Cooperman, L. Finkelstein, E. Luks, A. Seress, *Fast Monte Carlo Algorithms for Permutation Groups*, J. of Computer and System Sciences, vol. 50, nr. 2, pp. 296–308, Elsevier, 1995.
doi:10.1006/jcss.1995.1024

[Bau12] B. Baumeister, J.-H. de Wiljes, *Aperiodic Logarithmic Signatures*, J. of Math. Cryptology, vol. 6, nr. 1, pp. 21–37, de Gruyter, 2012.
doi:10.1515/jmc-2012-0003

[Bla09] S. R. Blackburn, C. Cid, C. Mullan, *Cryptanalysis of the MST_3 Public Key Cryptosystem*, J. of Math. Cryptology, vol. 3, nr. 4, pp. 321–338, de Gruyter, 2009.
doi:10.1515/JMC.2009.020

[Boh05] J.-M. Bohli, R. Steinwandt, M. I. González Vasco, C. Martínez, *Weak Keys in MST_1*, Des. Codes Cryptography, vol. 37, nr. 3, pp. 509–524, Kluwer/Springer, 2005.
doi:10.1007/s10623-004-4040-y

[Buc05] J. Buchmann, A. Schmidt, *Computing the Structure of a Finite Abelian Group*, Math. of Comp., vol. 74, nr. 252, pp. 2017–2026, American Mathematical Society, 2005.
doi:10.1090/S0025-5718-05-01740-0

[Car06] A. Caranti, F. D. Volta, *The Round Functions of Cryptosystem PGM Generate the Symmetric Group*, Des. Codes Cryptography, vol. 38, nr. 1, pp. 147–155, Kluwer/Springer, 2006.
doi:10.1007/s10623-005-5667-z

[Cus00] C. A. Cusack, *Group Factorizations in Cryptography*, Ph.D. dissertation, University of Nebraska, 2000.

[Fuc67] L. Fuchs, *Abelian Groups*, 3rd edition reprinted, Pergamon Press, 1967.

[Gol01] O. Goldreich, *Foundations of Cryptography: Volume I – Basic Tools*, Cambridge University Press, 2001.

[Hås99] J. Håstad, R. Impagliazzo, L. A. Levin, M. Luby, *A Pseudorandom Generator from any One-Way Function*, SIAM J. on Computing, vol. 28, nr. 4, pp. 1364–1396, 1999.
doi:10.1137/S0097539793244708

[Hol04] P. E. Holmes, *On Minimal Factorisations of Sporadic Groups*, Experiment. Math., vol. 13, nr. 4, pp. 435–440, A K Peters, 2004.
http://projecteuclid.org/euclid.em/1109106435

[Hup67] B. Huppert, *Endliche Gruppen I*, Springer, 1967.

[Ili85] C. S. Iliopoulos, *Analysis of Algorithms on Problems in General Abelian Groups*, Information Processing Letters, vol. 20, nr. 4, pp. 215–220, Elsevier, 1985.
doi:10.1016/0020-0190(85)90052-3

[Joh80] D. L. Johnson, *Topics in the Theory of Group Presentations*, London Mathematical Society Lecture Note Series, vol. 42, Cambridge University Press, 1980.

[Lem05] W. Lempken, T. van Trung, *On Minimal Logarithmic Signatures of Finite Groups*, Experiment. Math., vol. 14, nr. 3, pp. 257–269, A K Peters, 2005.
http://projecteuclid.org/euclid.em/1128371751

[Lem09] W. Lempken, T. van Trung, S. S. Magliveras, W. Wei, *A Public Key Cryptosystem Based on Non-Abelian Finite Groups*, J. of Cryptology, vol. 22, nr. 1, pp. 62–74, Springer, 2009.
doi:10.1007/s00145-008-9033-y

[Mag86] S. S. Magliveras, *A Cryptosystem from Logarithmic Signatures of Finite Groups*, in Proceedings of the 29th Midwest Symposium on Circuits and Systems, pp. 972–975, Elsevier, 1986.

[Mag92] S. S. Magliveras, N. D. Memon, *Algebraic Properties of Cryptosystem PGM*, J. of Cryptology, vol. 5, nr. 3, pp. 167–183, Springer, 1992.
doi:10.1007/BF02451113

[Mag02a] S. S. Magliveras, *Secret- and Public-Key Cryptosystems from Group Factorizations*, Tatra Mt. Math. Publ., vol. 25, pp. 11–22, 2002.

[Mag02b] S. S. Magliveras, D. R. Stinson, T. van Trung, *New Approaches to Designing Public Key Cryptosystems Using One-Way Functions and Trapdoors in Finite Groups*, J. of Cryptology, vol. 15, nr. 4, pp. 285–297, Springer, 2002.
doi:10.1007/s00145-001-0018-3

[Mar12] P. Marquardt, P. Svaba, T. van Trung, *Pseudorandom Number Generators Based on Random Covers for Finite Groups*, Des. Codes Cryptography, vol. 64, nr. 1–2, pp. 209–220, Springer, 2012.
doi:10.1007/s10623-011-9485-1

[Mot95] R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[Nus11] A. Nuss, *On Group Based Public Key Cryptography*, Ph.D. dissertation, University of Tübingen, 2011.
http://nbn-resolving.de/urn:nbn:de:bsz:21-opus-63659

[Poh78] S. C. Pohlig, M. E. Hellman, *An Improved Algorithm for Computing Logarithms over $GF(p)$ and its Cryptographic Significance*, IEEE Transactions on Information Theory, vol. 24, nr. 1, pp. 106–110, 1978.
doi:10.1109/TIT.1978.1055817

[Red65] L. Rédei, *Die neue Theorie der endlichen abelschen Gruppen und Verallgemeinerung des Hauptsatzes von Hajós*, Acta Math. Acad. Sci. Hung., vol. 16, nr. 3–4, pp. 329–373, Kluwer, 1965.
doi:10.1007/BF01904843

[Ser77] J. P. Serre, *Linear Representations of Finite Groups*, Graduate Texts in Mathematics, Springer, 1977.

[Sho99] P. W. Shor, *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*, SIAM J. on Computing, vol. 41, no. 2, pp. 303–332, 1999.
doi:10.1137/S0036144598347011

[Sta13] R. Staszewski, T. van Trung, *Strongly Aperiodic Logarithmic Signatures*, J. of Math. Cryptology, vol. 7, nr. 2, pp. 147–179, de Gruyter, 2013.
doi:10.1515/jmc-2013-5000

[Sva07] P. Svaba, T. van Trung, *On Generation of Random Covers for Finite Groups*, Tatra Mt. Math. Publ., vol. 37, pp. 105–112, 2007.

[Sva10] P. Svaba, T. van Trung, *Public Key Cryptosystem MST_3: Cryptanalysis and Realization*, J. of Math. Cryptology, vol. 4, no. 3, pp. 271–315, de Gruyter, 2010.
doi:10.1515/jmc.2010.011

[Sva11] P. Svaba, *Covers and Logarithmic Signatures of Finite Groups in Cryptography*, Ph.D. dissertation, University of Duisburg-Essen, 2011.
http://duepublico.uni-duisburg-essen.de/servlets/DocumentServlet?id=25967

[Sva13] P. Svaba, T. van Trung, P. Wolf, *Logarithmic Signatures for Abelian Groups and Their Factorization*, Tatra Mt. Math. Publ., vol. 57, no. 1, pp. 21–33, 2013.
doi:10.2478/tmmp-2013-0033

[Sza04] S. Szabó, *Topics in Factorization of Abelian Groups*, Birkhäuser, 2004.

[Sza09] S. Szabó, A. D. Sands, *Factoring Groups into Subsets*, Lecture Notes in Pure and Applied Mathematics, vol. 257, Chapman and Hall / CRC Press, 2009.

[Vas02] M. I. González Vasco, R. Steinwandt, *Obstacles in Two Public Key Cryptosystems Based on Group Factorizations*, Tatra Mt. Math. Publ., vol. 25, pp. 23–37, 2002.

[Vas03] M. I. González Vasco, M. Rötteler, R. Steinwandt, *On Minimal Length Factorizations of Finite Groups*, Experiment. Math., vol. 12, nr. 1, A K Peters, 2003. http://projecteuclid.org/euclid.em/1064858780

[Vas04] M. I. González Vasco, D. Hofheinz, C. Martínez, R. Steinwandt, *On the Security of Two Public Key Cryptosystems Using Non-Abelian Groups*, Des. Codes Cryptography, vol. 32, nr. 1–3, pp. 207–216, Kluwer/Springer, 2004. doi:10.1023/B:DESI.0000029223.76665.7e

[Vas10] M. I. González Vasco, A. L. Pérez del Pozo, P. Taborda Duarte, *A Note on the Security of MST_3*, Des. Codes Cryptography, vol. 55, nr. 2–3, pp. 189–200, Springer, 2010. doi:10.1007/s10623-010-9373-0

List of Abbreviations

The following table lists abbreviations used throughout this work.

Abbreviation	Meaning
AES-256	Advanced Encryption Standard with 256-bit key (a symmetric block cipher, considered as being secure today).
CBC	Cipher Block Chaining mode (block cipher mode of operation).
CSPRNG	Cryptographically Secure Pseudo-Random Number Generator.
CTR	Counter mode (block cipher mode of operation).
DLL	Dynamic Link Library.
DLP	Discrete Logarithm Problem.
ECB	Electronic Code Book mode (block cipher mode of operation).
GUI	Graphical User Interface.
HTTPS	Hypertext Transfer Protocol Secure.
ILP	Integer Linear Programming.
IO	Input/Output.
IV	Initialization Vector used in several block cipher modes of operation, e.g. CBC.
LFSR	Linear Feedback Shift Register.
PGM	Permutation Group Mappings.
PKCS	Public-Key Cryptography Standard.
PRNG	Pseudo-Random Number Generator.
S/MIME	Secure/Multipurpose Internet Mail Extensions.
SCP	Secure Copy.
SHA-256	256-bit version of the Secure Hash Algorithm (a cryptographic hash algorithm, considered as being secure today).
SSH	Secure Shell.
SSL	Secure Sockets Layer.
TLS	Transport Layer Security.
TSN	Translation, Element Shuffle and Normalization.
UI	User Interface.
UTF-8	8-bit Unicode Transformation Format (mapping of Unicode characters to byte sequences).
W.l.o.g.	Without loss of generality.
XML	Extensible Markup Language (markup language for representing hierarchically structured data in text form).

Index to Notations

The following table lists symbols used throughout this work.

Symbolism	Meaning	Where Def.
$\square$	End of proof.	
$\exists\, x$	At least one x exists.	
$\exists!\, x$	Exactly one x exists.	
$\nexists\, x$	No x exists.	
$\forall\, x$	For all x.	
$\emptyset$	Empty set.	
$\lfloor r \rfloor$	Largest $z \in \mathbb{Z}$ with $z \leq r$.	
$\lceil r \rceil$	Smallest $z \in \mathbb{Z}$ with $z \geq r$.	
$\lvert G \rvert$	Order of the group G.	2.1
$\langle g \rangle$	Cyclic subgroup generated by the group element g, $\langle g \rangle := \{g^n \mid n \in \mathbb{Z}\}$.	2.1
$\langle S \rangle$	Smallest subgroup of G containing the set $S \subseteq G$.	2.1
G^n	Direct product/sum of n copies of the group G.	2.1
$[x, y]$	Closed interval, all values between x and y (including x and y).	
$]x, y[$	Open interval, all values between x and y (excluding x and y).	
$u \leftarrow v$	In pseudo-code: set the variable u to the value v.	2.8
$G \times H$	Direct product of G and H.	2.1
$G \oplus H$	Direct sum of the abelian G and H.	2.1
$N \rtimes_\varphi H$	Semidirect product of the groups N and H with respect to the group homomorphism $\varphi : H \to \mathrm{Aut}(N)$.	2.1
$G \cong H$	G and H are isomorphic.	2.1
$U \leq G$	U is a subgroup of the group G.	2.1
$N \trianglelefteq G$	N is a normal subgroup of the group G.	2.1
$x \approx y$	x is approximately y.	
$x \gg y$	x is far larger than y.	
$f_2 \circ f_1$	Composition of the functions f_1 and f_2, $(f_2 \circ f_1)(x) = f_2(f_1(x))$.	
$x \mid y$	x divides y without remainder.	
$x \equiv y \pmod m$	Congruence relation (on $\mathbb{Z}$), $x \equiv y \pmod m \Leftrightarrow m \mid x - y$.	

Symbolism	Meaning	Where Def.
$x \bmod y$	Modulo function, $x \bmod y := \min\{n \in \mathbb{N}_0 \mid n \equiv x \pmod{y}\}$.	
$\alpha \breve{=} \beta$	For $\alpha, \beta \in \Xi(G)$, $\alpha \breve{=} \beta \Leftrightarrow \breve{\alpha} = \breve{\beta}$.	3.2
$\mathfrak{a}_\varphi(\alpha)$	Automorphism application on $\alpha \in \Xi(G)$, $\mathfrak{a}_\varphi : \Xi(G) \to \Xi(G) : \alpha \mapsto \varphi(\alpha)$.	5.1.10
$\mathfrak{A}(G)$	Group of all automorphism application transformations, $\mathfrak{A}(G) := \{\mathfrak{a}_\varphi \mid \varphi \in \mathrm{Aut}(G)\}$.	5.1.10
$\mathrm{Alt}(n)$	Alternating group on the set $\{1, 2, \ldots, n\}$.	2.3.1
$\mathrm{Alt}(X)$	Alternating group on the set X.	2.3.1
$\mathcal{AT}(G)$	Set of all amalgamated transversal logarithmic signatures for the group G.	6.3
$\mathrm{Aut}(G)$	Group of automorphisms on the group G.	2.1
$\mathrm{Aut}_c(G)$	Group of central automorphisms on the group G, $\mathrm{Aut}_c(G) := \{\varphi \in \mathrm{Aut}(G) \mid g^{-1}\varphi(g) \in Z(G) \text{ for all } g \in G\}$.	2.1
$\mathbb{C}$	Complex numbers.	
$C_G(S)$	Centralizer of the subset $S \subseteq G$, $C_G(S) := \{g \in G \mid sg = gs \text{ for all } s \in S\} \leq G$.	2.1
$\mathfrak{c}_g(A)$	Multiplicity of g in the block A.	2.4
$\mathsf{c}_e(A)$	$\mathsf{c}_e : \mathcal{P}(G) \to \mathbb{N}_0 : A \mapsto \lvert\{\pi \in A \mid \mathrm{sgn}(\pi) = 1\}\rvert$.	2.3.1
$\mathsf{c}_o(A)$	$\mathsf{c}_o : \mathcal{P}(G) \to \mathbb{N}_0 : A \mapsto \lvert\{\pi \in A \mid \mathrm{sgn}(\pi) = -1\}\rvert$.	2.3.1
$\mathscr{c}_\sigma(A)$	$\mathscr{c}_\sigma : \mathcal{P}(G) \to \mathbb{N}_0 : A \mapsto \lvert\{g \in A \mid g \text{ is a rotation}\}\rvert$.	2.3.3.1
$\mathscr{c}_\tau(A)$	$\mathscr{c}_\tau : \mathcal{P}(G) \to \mathbb{N}_0 : A \mapsto \lvert\{g \in A \mid g \text{ is a reflection}\}\rvert$.	2.3.3.1
$\acute{\chi}(A)$	For a character $\chi : G \to \mathbb{C}$ and $A \subseteq G$: $\acute{\chi}(A) := \sum_{g \in A} \chi(g)$.	5.3
$\mathrm{Cl}(g)$	Conjugacy class of a $g \in G$, $\mathrm{Cl}(g) := \{hgh^{-1} \mid h \in G\}$.	2.1
D_{2n}	Dihedral group of order $2n$.	2.3.3.1
$\mathfrak{D}_{2^n}$	Quasi-dihedral group of order 2^n.	2.3.5
$\delta_{i,j}$	Kronecker delta, $\delta_{i,j} := \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases}$	
$E(\alpha)$	Set of all elements in the blocks of $\alpha \in \Xi_{(r_1,\ldots,r_s)}(G)$, $E(\alpha) := \bigcup_{i=1}^{s} \{\alpha[i][j] \mid 1 \leq j \leq r_i\}$.	3.1
$E(A)$	Set of all elements in the block A, $E(A) := \{A[i] \mid 1 \leq i \leq \lvert A\rvert\}$.	3.1
$\mathcal{E}(G)$	Set of all exact transversal logarithmic signatures for the group G.	3.4.3
$\mathfrak{e}_{(\pi_1,\ldots,\pi_s)}(\alpha)$	Element shuffle of $\alpha \in \Xi_{(r_1,\ldots,r_s)}(G)$ using the permutations $\pi_1 \in \mathrm{Sym}(r_1), \ldots, \pi_s \in \mathrm{Sym}(r_s)$.	5.1.1

Symbolism	Meaning	Where Def.
$\mathfrak{E}_{(r_1,\ldots,r_s)}(G)$	Group of all element shuffle transformations, $\mathfrak{E}_{(r_1,\ldots,r_s)}(G) := \{\mathfrak{e}_{(\pi_1,\ldots,\pi_s)} \mid \pi_1 \in \mathrm{Sym}(r_1), \ldots, \pi_s \in \mathrm{Sym}(r_s)\}$.	5.1.1
$\mathbb{F}_q$	Finite field of order q, where $q = p^n$ with $p \in \mathbb{P}, n \in \mathbb{N}$.	
$\mathfrak{G}_{p \cdot p^n}$	$\mathfrak{G}_{p \cdot p^n} := \left\langle \sigma, \tau \mid \sigma^{p^n} = \tau^p = \mathrm{id}, \tau^{-1}\sigma\tau = \sigma^{1+p^{n-1}} \right\rangle$.	2.3.5
$\gcd(x, y)$	Greatest common divisor of x and y.	
$\mathrm{GL}(n, K)$	General linear group of degree n over K, $\mathrm{GL}(n, K) := \{M \in \mathcal{M}_n(K) \mid M \text{ is invertible}\}$.	2.2
$\mathrm{im}(f)$	Image of the mapping f.	2.1
$\mathrm{Inn}(G)$	Group of inner automorphisms on the group G, $\mathrm{Inn}(G) := \{\varphi : G \to G : x \mapsto g^{-1}xg \mid g \in G\}$.	2.1
$\ker(f)$	Kernel of the mapping f.	2.1
$\ell(\alpha)$	Length of $\alpha \in \Xi_{(r_1,\ldots,r_s)}(G)$, $\ell(\alpha) := \sum_{i=1}^{s} r_i$.	3.1
$\Lambda(G)$	Set of all logarithmic signatures for the group G.	3.1
$\mathrm{lcm}(x, y)$	Least common multiple of x and y.	
$\ln x$	Natural logarithm of x (with respect to base e).	
$\log x$	Logarithm of x with respect to base 10.	
$\log_r x$	Logarithm of x with respect to base r.	
$\mathcal{M}_n(S)$	Set of all $n \times n$ matrices over the set S.	
$\mathfrak{M}_{(r_1,\ldots,r_s)}(G)$	$\mathfrak{M}_{(r_1,\ldots,r_s)}(G) := \mathfrak{A}(G) \circ \mathfrak{E}_{(r_1,\ldots,r_s)}(G) \circ \mathfrak{S}_s(G) \circ \mathfrak{T}(G)$.	5.1.14
$\mathbb{N}$	Natural numbers, $\mathbb{N} := \{1, 2, 3, \ldots\}$.	
$\mathbb{N}_0$	Natural numbers with 0, $\mathbb{N}_0 := \{0, 1, 2, \ldots\}$.	
$\mathbb{N}_{\geq k}$	Natural numbers greater than or equal to k, $\mathbb{N}_{\geq k} := \{n \in \mathbb{N}_0 \mid n \geq k\}$.	
$\mathcal{NT}(G)$	Set of all logarithmic signatures for the group G that are not transversal, $\mathcal{NT}(G) = \Lambda(G) \setminus \mathcal{T}(G)$.	3.4.3
$O(f)$	Set of all functions with an absolute asymptotic growth rate less or equal to the one of the function f, $O(f) := \{g : x \mapsto g(x) \mid \exists\, c > 0, x_0 : \lvert g(x)\rvert \leq c \cdot \lvert f(x)\rvert \text{ for all } x \geq x_0\}$.	
$\Omega(f)$	Set of all functions with an absolute asymptotic growth rate greater or equal to the one of the function f, $\Omega(f) := \{g : x \mapsto g(x) \mid \exists\, c > 0, x_0 : \lvert g(x)\rvert \geq c \cdot \lvert f(x)\rvert \text{ for all } x \geq x_0\}$.	
$\mathrm{ord}(g)$	Order of g, $\mathrm{ord}(g) := \lvert\langle g\rangle\rvert$.	2.1
$\mathbb{P}$	Prime numbers in $\mathbb{N}$.	
$\mathbb{P}_{\geq k}$	Prime numbers greater than or equal to k, $\mathbb{P}_{\geq k} := \{p \in \mathbb{P} \mid p \geq k\}$.	
$\mathcal{P}(S)$	Power set of S, $\mathcal{P}(S) := \{U \mid U \subseteq S\}$.	
$\Pr[\mathfrak{E}]$	Probability that event $\mathfrak{E}$ occurs.	
$\mathbb{Q}$	Rational numbers.	

Symbolism	Meaning	Where Def.
Q_{4n}	Generalized quaternion group of order $4n$.	2.3.4
$\mathbb{R}$	Real numbers.	
$\mathrm{rank}(M)$	Maximum number of linear independent row/column vectors of the matrix M.	
$\acute{\rho}(A)$	For a $\mathbb{C}$-representation $\rho : G \to \mathrm{GL}(n,\mathbb{C})$ and a subset or multiset $A \subseteq G$: $\acute{\rho}(A) := \sum_{g\in A} \rho(g)$.	5.3
$\mathcal{S}(G,\alpha)$	$\mathcal{S}(G,\alpha) := b_G \cdot \ell(\alpha)$, where b_G is the code length.	3.3
$\mathfrak{s}_{(x_1,\ldots,x_{s-1})}(\alpha)$	Sandwich of $\alpha \in \Xi_s(G)$ using $x_1,\ldots,x_{s-1} \in G$, $\mathfrak{s}_{(x_1,\ldots,x_{s-1})} : \Xi_s(G) \to \Xi_s(G) : \alpha = (A_1,\ldots,A_s) \mapsto (A_1x_1, x_1^{-1}A_2x_2, \ldots, x_{s-1}^{-1}A_s)$.	5.1.4
$\mathfrak{S}_s(G)$	Group of all sandwich transformations, $\mathfrak{S}_s(G) := \{\mathfrak{s}_{(x_1,\ldots,x_{s-1})} \mid x_1,\ldots,x_{s-1} \in G\}$.	5.1.4
$\mathrm{sgn}(\pi)$	Sign of the permutation π, $\mathrm{sgn}(\pi) := (-1)^{N(\pi)}$ with $N(\pi)$ the number of inversions in π.	2.3.1
$\mathrm{Syl}_p(G)$	Sylow p-subgroup of the group G.	
$\mathrm{Sym}(n)$	Symmetric group on the set $\{1,2,\ldots,n\}$.	2.3.1
$\mathrm{Sym}(X)$	Symmetric group on the set X.	2.3.1
$t(\alpha)$	Type of the block sequence $\alpha \in \Xi_{(r_1,\ldots,r_s)}(G)$, $t(\alpha) := (r_1,\ldots,r_s)$.	3.1
$t(G)$	Type of the abelian group G.	2.1
$\mathcal{T}(G)$	Set of all transversal logarithmic signatures for the group G.	3.4.3
$\mathfrak{t}_{x,y}(\alpha)$	Translation of $\alpha \in \Xi(G)$ by $x,y \in G$, $\mathfrak{t}_{x,y} : \Xi(G) \to \Xi(G) : \alpha \mapsto x\alpha y$.	5.1.3
$\mathfrak{T}(G)$	Group of all translation transformations, $\mathfrak{T}(G) := \{\mathfrak{t}_{x,y} \mid x,y \in G\}$.	5.1.3
$\Theta(f)$	Set of all functions with the same absolute asymptotic growth rate as the function f, $\Theta(f) := O(f) \cap \Omega(f)$.	
Θ_α	Element selection/product mapping.	3.2
$\mathcal{TNT}(G)$	Set of all logarithmic signatures for G where no block is a coset of a subgroup of G.	3.4.3
$\mathrm{tr}(M)$	Trace of the square matrix M, $\mathrm{tr} : \mathcal{M}_n(K) \to K : M \mapsto \sum_{i=1}^{n} M_{i,i}$.	
$\Xi(G)$	Set of all finite block sequences of elements of G.	3.1
$\Xi_s(G)$	$\Xi_s(G) := \{(A_1,\ldots,A_m) \in \Xi(G) \mid m = s\}$.	3.1
$\Xi_{(r_1,\ldots,r_s)}(G)$	$\Xi_{(r_1,\ldots,r_s)}(G) := \{\alpha \in \Xi_s(G) \mid t(\alpha) = (r_1,\ldots,r_s)\}$.	3.1
$\mathbb{Z}$	Integers, $\mathbb{Z} := \{\ldots,-2,-1,0,1,2,\ldots\}$.	
$\mathbb{Z}_n$	Cyclic group $(\mathbb{Z}/n\mathbb{Z},+)$ of order n, factor ring $(\mathbb{Z}/n\mathbb{Z},+,\cdot)$.	2.3.2

Symbolism	Meaning	Where Def.
$\mathbb{Z}_n^k$	Direct sum of k copies of $\mathbb{Z}_n$, $\mathbb{Z}_n^k = \underbrace{\mathbb{Z}_n \oplus \mathbb{Z}_n \oplus \ldots \oplus \mathbb{Z}_n}_{k \text{ times}}$.	2.3.2
$\mathbb{Z}[x]$	Set of all polynomials over $\mathbb{Z}$.	
$Z(G)$	Center of the group G, $Z(G) := \{z \in G \mid zg = gz \text{ for all } g \in G\}$.	2.1
$\mathfrak{Z}(R)$	$\mathfrak{Z} : \mathcal{P}(G) \to \mathcal{P}(G) : R \mapsto \{z \in R \mid z \cdot R^{-1} = R\}$.	9.1.2
$\widehat{\mathfrak{Z}}(R)$	$\widehat{\mathfrak{Z}} : \mathcal{P}(G) \to \mathcal{P}(G) : R \mapsto \mathfrak{Z}(R) \cdot \mathfrak{Z}(R)^{-1}$.	9.1.2

Index