Optimal Merging in Quantum k-xor and k-sum Algorithms

María Naya-Plasencia and André Schrottenloher

Inria, France
(maria.naya_plasencia,andre.schrottenloher)@inria.fr

Abstract. The k-xor problem or Generalized Birthday Problem asks, given k lists of bit-strings, for a k-tuple among them XORing to 0. If the lists are unbounded, the best classical (exponential) time complexity has withstood since Wagner's CRYPTO 2002 paper, while many subsequent works have improved its memory complexity, given better time-memory tradeoffs, and logarithmic improvements.
In this paper, we study quantum algorithms for several variants of the k-xor problem. When quantum oracle access is allowed, we improve over previous results of Grassi *et al.* for almost all values of k. We define a set of "merging trees" which represent strategies for quantum and classical merging in k-xor algorithms, and prove that our method is optimal among these. We provide, for the first time, quantum speedups when the lists can be queried *only classically*.
We also extend our study to lists of limited size, up to the case where a single solution exists. We give quantum dissection algorithms that outperform the best known for many k, and apply to the multiple-encryption problem. Our complexities are confirmed by a Mixed Integer Linear Program that computes the best strategy for a given k-xor problem. All our algorithms apply when considering modular additions instead of bitwise XORs.

Keywords: generalized birthday problem, quantum cryptanalysis, list-merging algorithms, k-list problems, multiple encryption, MILP.

1 Introduction

As constant progress is being made in the direction of quantum computing devices with practical applications, the inherent threat to cryptography has led to massive amounts of research in designing secure post-quantum primitives. To design these cryptosystems and justify their parameters, one must rely on generic levels of quantum security. Therefore a precise study of the *query* and *time* complexities of quantum algorithms for relevant problems is needed. Furthermore, improved quantum algorithms may increase the vulnerabilities of some cryptosystems. In this work, we study, from a quantum point of view, an ubiquitous generic problem with many variants and applications: the Generalized Birthday Problem, or k-xor problem.

Generalized Birthday Problem. The birthday problem, or collision problem, may be formulated as the following: given a random oracle $H : \{0,1\}^n \to \{0,1\}^n$, find a collision pair, *i.e.* $x, y \in \{0,1\}^n$ such that $H(x) = H(y)$. The function H may model an oracle access to a cryptographic primitive (such as a hash function) behaving as ideal, and the hardness of this problem defines its generic security. When using only classical queries and computations, it is well-known that $\Omega(2^{n/2})$ queries are necessary and sufficient to retrieve a collision.

In a seminal paper, Wagner [32] generalized a method credited to Camion and Patarin [13] to solve a generalization of this problem to k-tuples. There are different possible formulations, which may impact the algorithms used. In general, if we consider no oracle access, the problem is:

> Given some lists $L_1, \ldots, L_k$ of n-bit strings, find a k-tuple $x_1, \ldots x_k$ of $L_1 \times \ldots \times L_k$ such that $x_1 \oplus x_2 \oplus \ldots \oplus x_k = 0$.

Extension to Other Operations. The bitwise XOR operation $\oplus$ is natural in many cryptographic applications. In Wagner's algorithm [32], its classical variants, and [17], it can be replaced by modular additions. We focus on $\oplus$ but will justify later that all our results apply to $+$.

Classical Complexity of k-xor. Intuitively, starting from $k = 2$ (collisions), increasing k can only make the problem easier on average, since new degrees of freedom are available. The query complexity of k-xor is $\widetilde{\Omega}(2^{n/k})$ queries: with them it is possible to build $\mathcal{O}(2^n)$ k-tuples, and retrieve a XOR to zero with constant probability. The main contribution of Wagner in [32] is to give an algorithm which, although far from optimal in queries, reaches an efficient time complexity for any k. Its time complexity is $\widetilde{\mathcal{O}}\left(2^{n/(\lfloor \log_2(k) \rfloor + 1)}\right)$, using k lists of size $2^{n/(\lfloor \log_2(k) \rfloor + 1)}$. If we disregard logarithmic factors, as will be done in the remaining of this paper, we observe that the exponent decreases only when k encounters a new power of 2. For any ℓ, $k = 2^\ell$ and $k = 2^{\ell+1} - 1$ have the same exponent.

Quantum k-xor. The situation in the quantum setting might seem, at first sight, not so different. The quantum query complexity of k-xor is known to be $\Omega\left(2^{n/(k+1)}\right)$. In [17] some quantum algorithms for the solving the k-xor problem with quantum oracle access are given. For a general k, the time complexity of $\widetilde{\mathcal{O}}\left(2^{n/(\lfloor \log_2(k) \rfloor + 2)}\right)$ is obtained in the MNRS quantum walk framework [25]. It also decreases at each new power of 2. One of the observations in [17] is that, contrary to the classical state-of-the-art, there exists an exponential separation between the quantum collision and 3-xor time complexities. While collision search requires provably $\Omega(2^{n/3})$ quantum queries, the authors present a 3-xor algorithm running in time $\mathcal{O}\left(2^{3n/10}\right)$. They ask whether other non-powers of 2 have better quantum algorithms and if time speedups with only $poly(n)$ qubits, obtained in some cases, are reachable for all k.

This paper. In this work, we first answer these open questions by introducing "merging trees". These trees describe in a systematic way merging strategies to solve the quantum k-xor problem. This enables us to reach better exponential time complexities than [17], with exponents that decrease strictly at each new value of k (we also improve their 3-xor algorithm with qRAM). With $poly(n)$ qubits and without qRAM, we give quantum speedups for half of the values of k. We prove that our results are optimal among all merging trees.

Furthermore, we extend our framework to the k-xor problem with classically given lists, while [17] only studied the problem with quantum oracle access, and obtain an effective quantum speedup. We also extend our algorithms to lists of limited size, up to the case where k lists of size only $2^{n/k}$ are given as input, in which we improve the current best algorithms for most values of k. In addition, we show how all the algorithms also apply in the k-sum case, and we provide some examples of applications. The main algorithmic results are summarized in Section 3.

Outline. The remaining of this paper is organized as follows. In Section 2, we recall some preliminaries of quantum computing, state the different problems that we will solve and recall previous results. Section 3 summarizes our main results. In Section 4, we present Wagner's algorithm and show how to generalize its idea with the concept of *merging trees*, which can be adapted to the quantum setting. These strategies cover all the previously known quantum algorithms for k-xor and the new ones in this paper. Our results were first obtained experimentally with the help of Mixed Integer Linear Programming, as the complexity of a merging tree appears naturally as the solution to a simple linear optimization problem. This is why our definition focuses on *variables* and *constraints*. In Section 5, we give the optimal merging trees for quantum k-xor and prove their optimality among all strategies of our framework. We also compare our new results with the ones from [17]. Next, in Section 6, we extend to limited input domains, *i.e.* smaller lists, to classical inputs and to the multiple-encryption problem. Finally, in Section 7, we provide some applications. We conclude the paper with some open questions.

2 Preliminaries

In this section we introduce the problems under study, cover some basic required notions of quantum computing and summarize the state-of-the-art of algorithms for these k-xor problems.

2.1 Variants of the k-xor Problems

All algorithms in this paper have exponential time complexities in n, written $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$ for some α_k depending *only* on k. We consider k as a constant and neglect the multiplicative factors in k and n.

The k-xor problem has two main variants: the input data can be accessed via *input lists* or via an *oracle*. Classically, this does not make any (more than constant in k) difference. Quantumly, it implicitly determines whether we authorize *quantum access* or only classical access to the data.

Problem 1 (k-xor with lists). Given $L_1, \dots L_k$ lists of random n-bit strings, find $x_1, \dots x_k \in L_1 \times \dots L_k$ such that $x_1 \oplus \dots \oplus x_k = 0$ in minimal time.

Problem 1 is the original problem solved by Wagner in [32]. The goal is to obtain the best time complexity knowing that the sizes of the list is arbitrary, and not a concern. Notice that in that case, there exists an optimal list size, which is exponential in n (otherwise we wouldn't expect a solution) and the same for all lists (otherwise we could increase the size of the non-maximal lists and simply drop the additional elements).

In the *oracle* version of this problem, the oracle has unrestricted domain, and can produce up to 2^n elements.

Problem 2 (k-xor with an oracle). Given oracle access to a random n-bit to n-bit function H, find $x_1, \dots x_k \in L_1 \times \dots L_k$ such that $H(x_1) \oplus \dots \oplus H(x_k) = 0$.

Alternatively, one can define Problem 2 with k different random functions, or Problem 1 with a single input list. These formulations are equivalent up to a constant factor in k, and they will be of less importance in the rest of this paper.

Problem 2 is the one studied in [17], when quantum oracle access to H is allowed. In that case, instead of querying H for a fixed input x, we are allowed superposition queries to a quantum oracle O_H. This models a situation in which the production of the lists is entirely controlled by the adversary, and can be easily implemented on a quantum computer. However, this does not encompass all scenarios. In this paper, we will also study Problem 1, with a purely classical input data.

Moreover, during our study, we will allow a limitation of the domain of H, or alternatively, of the sizes of the lists L_i. The limit case happens when there is on average a single k-tuple with a XOR to zero. We name these problems "unique k-xor" with an oracle or with lists.

Problem 3 (Unique k-xor with an oracle). Given query access to a random $\lceil n/k \rceil$-bit to n-bit function H, expecting that there exists a single k-tuple $x_1, \dots x_k$ such that $H(x_1) \oplus H(x_2) \oplus \dots H(x_k) = 0$, find it.

Although we choose to focus on these limit cases, our framework will encompass all intermediate cases where the domain size of H (or the size of L_i) is restricted to 2^d with $\lceil \frac{n}{k} \rceil \leq d \leq n$.

Problem 4 (Unique k-xor with lists). Given classical data as k lists $L_1, \dots, L_k$ of n-bit strings, of size $2^{n/k}$, find a k-tuple $x_1, \dots x_k \in L_1 \times \dots \times L_k$ such that $x_1 \oplus \dots \oplus x_k = 0$, if it exists.

2.2 Quantum Computing Model and Preliminaries

We design generic algorithms in the quantum circuit model. However, as we are only interested in exponential time complexities, we allow ourselves a level of abstraction which should make our algorithms and complexities understandable even for a non-expert audience. For the interested reader, a thorough introduction to quantum computing can be found in [29].

The quantum circuit model is a universal way of describing a quantum computation. We compute with a set of qubits, which are two-dimensional quantum systems. Their state is described by a vector in a Hilbert space $\mathcal{H}$, of the form $\alpha|0\rangle + \beta|1\rangle$, where $|0\rangle, |1\rangle$ is the canonical basis of $\mathcal{H}$ (named the computational basis), α, β are complex numbers and $|\alpha|^2 + |\beta|^2 = 1$. A quantum circuit starts with a system of (possibly many) qubits in the state $|0\rangle$; then a sequence of computations (*quantum gates*), possibly interleaved with oracle calls, is applied. At the end of the computation, the qubits are measured. Measurement is destructive: it allows to extract information from a superposition, but causes it to *collapse*. For example, measuring $\alpha|0\rangle + \beta|1\rangle$ gives 0 with probability $|\alpha|^2$ while setting the state to $|0\rangle$.

A widely known example of quantum algorithm is Grover's algorithm [19].

Lemma 1 (Grover Search, from [19]). *Let X be a search space, whose elements are represented on $\lceil \log_2(|X|) \rceil$ qubits, such that the uniform superposition $\frac{1}{\sqrt{|X|}} \sum_{x \in X} |x\rangle$ is computable in $\widetilde{\mathcal{O}}(1)$ time. Let $f : X \to \{0,1\}$ be a "testing function", such that we can implement a superposition oracle $O_f(|x\rangle|b\rangle) = |x\rangle|b \oplus f(x)\rangle$ in $\widetilde{\mathcal{O}}(1)$ time. Let $G \subset X$ be the subset of "good" elements $x \in X$ such that $f(x) = 1$. Then there exists a quantum algorithm using $\lceil \log_2(|X|) \rceil$ qubits, running in time $\widetilde{\mathcal{O}}\left(\sqrt{|X|/|G|}\right)$ that returns some $x \in G$. In particular, if $|G| = 1$, the running time is $\widetilde{\mathcal{O}}\left(\sqrt{|X|}\right)$.*

Building the superposition of all n-bit strings $\sum_{x \in \{0,1\}^n} |x\rangle$ is easy using n Hadamard gates. This is our basic search space in this paper.

Amplitude Amplification. In [10] a generalization of Grover was presented that enables to run a search with a structured search space: if there are 2^t partial solutions amongst the search space X, and if the superposition of elements of X can be constructed with a quantum algorithm $\mathcal{A}$ of complexity $|\mathcal{A}|$, we can recover the superposition of all preimages of 1 with total time $\widetilde{\mathcal{O}}\left(\sqrt{|X|2^{-t}}(|\mathcal{A}| + |O_f|)\right)$. As this procedure consists in iterating $\sqrt{|X|2^{-t}}$ times the same operator, we speak of "iterations".

In the rest of this paper, we use Grover search as a subroutine. We perform sequences of Grover searches, and also, nested instances, using Amplitude Amplification. We do the complexity estimates as if Grover's algorithm ran in exact time $\sqrt{|X|/|G|}$ and with success probability 1. More justification is provided in Appendix A.1.

Benchmarking. We focus on the single-processor model, and count the asymptotic *quantum time* complexity (the number of gates in the circuit), *quantum space* complexity (the number of qubits in the circuit) and, when necessary, classical time and space. This is contrary to works which focus primarily on quantum *query complexity* (*e.g.* [22]), or detailed quantum gate counts [2,18]. When an oracle is given, we consider oracle calls in time $\mathcal{O}(1)$ and suppose a constant quantum space overhead. Asymptotically, we consider that one quantum gate is equivalent to one classical gate. In practice, there should be a massive (but constant) factor in-between.

Scenario with or without qRAM. Some quantum algorithms for generic cryptographic applications, such as collision-finding [11] or unique collision-finding [1], yet optimal in queries and time, use massive amounts of *quantum random-access memory*. Much like classical random-access memory with constant-time access to memory cells decided at runtime, quantum RAM authorizes *superposition* access to its contents. Such a qRAM can be modeled by "qRAM gates", an add-on to a traditional universal gate set. Assume that the quantum circuit holds qubit registers $x_0 \dots x_{2^n-1}$. Then on input:

$$\left(\bigotimes_{j\in\{0,1\}^n} |x_j\rangle\right) \otimes |i\rangle\,|0\rangle \text{ we compute } \left(\bigotimes_{j\in\{0,1\}^n} |x_j\rangle\right) \otimes |i\rangle\,|x_i\rangle$$

in a single time step, realizing *superposition access* to the qubit registers. qRAM is useful to obtain good quantum time complexities, but its physical realization seems an order of magnitude harder than "baseline" universal quantum computation. This is why, following [17], we will study two scenarios: one considering quantum algorithms running without qRAM, and using only $\mathcal{O}(n)$ qubits ("low-qubits" case) and another with qRAM. Our algorithms for solving Problems 1, 4 and 3 will require qRAM for being efficient, while Problem 2 will be interesting in both scenarios.

2.3 Previous Algorithms for the k-xor Problem

Classical Algorithms for the k-xor Problem. In Section 4, we will describe in detail the algorithm of [32], that provides the current best classical exponential time complexity of $\widetilde{\mathcal{O}}\left(2^{n/(\lfloor \log_2(k)\rfloor+1)}\right)$ for any k (there are logarithmic improvements for non-powers of 2).

There have been many works regarding time-memory tradeoffs in Wagner's algorithm, for example [6,30]. In [26], the authors write linear programs to obtain the best strategy when the size of the lists is limited (equivalent to an oracle H with limited input domain).

Later on, more generalized frameworks have appeared, like [15], in which Dinur gives a memory improvement for some values of k and better time-memory tradeoffs in general, by combining parallel collision search, which is used in [30], with dissection [31,16]. We should mention [3] in which the authors study the

Short Integer Solution problem, and in this context, take a global view of generalized birthday and subset-sum algorithms based on merging lists.

Quantum Algorithms for the k-xor Problem. The first algorithm to find quantum collisions was found by Brassard, Høyer and Tapp in 1998 [12,11]. With a random function $H : \{0,1\}^n \to \{0,1\}^n$, it runs in time $\widetilde{\mathcal{O}}\left(2^{n/3}\right)$, using as much quantum queries. The bound $\Omega\left(2^{n/3}\right)$ was later proven to be optimal.

When a single collision is to be found, *i.e.* all elements are distinct, except two of them, Ambainis' algorithm [1], one of the most cited examples of a *quantum walk*, applies. Both these algorithms run in the qRAM model, requiring (very) big quantum hardware, whose issue was raised by Grover and Rudolph [20].

In ASIACRYPT 2017, Chailloux *et al.* [14] showed that it is possible to reach a total number of computations of $\mathcal{O}\left(2^{2n/5}\right)$ for collision search without qRAM. The search for a distinguished collision, instead of a general one, causes little time overhead while ensuring that the qRAM queries can actually be replaced by sequential lookups of a classical storage.

Recently, quantum algorithms for solving the k-xor problem were proposed in [17]. They were based on ideas from [32], [14] and [28].

3 Summary of our Main Results

In this section we summarize our main results: the optimal complexities, *in our merging tree framework*, for solving Problems 1, 2, 3 and 4, with XORs and modular additions, which allow to improve the quantum algorithms for a wide range of merging problems. The details will be given in the following sections.

The origin of this work was realizing that for some values of k, we were able to obtain merging algorithms that were more efficient than the ones from [17]. This could be done by decomposing the original k-xor problem on n bits in smaller problems, with smaller values of k' and a smaller number of bits, and merging them together. At the beginning, we did not find an intuitive way to predict the best merging strategies for a given k. We decided to implement a Mixed Integer Linear program[a] that gave us the best possible algorithms for many values of $k \leq 20$. From these results, we were able to understand the optimal methods. We extrapolated the theorems and propositions summarized here, that determine the optimal exponents in the complexities.

3.1 Quantum Algorithms for Problem 2

In the qRAM setting, we prove the following.

Theorem 1. *Let $k \geq 2$ be an integer and $\kappa = \lfloor \log_2(k) \rfloor$. The best quantum merging tree finds a k-xor on n bits in quantum time and memory $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$*

[a] Our code is available at `https://project.inria.fr/quasymodo/files/2019/05/merging_kxor_eprint.tar.gz`

where $\alpha_k = \frac{2^\kappa}{(1+\kappa)2^\kappa + k}$. *For* $c \leq 1$, *the same method finds* 2^{nc} *k-xor with a quantum (time and memory) complexity exponent of* $n \max(\alpha_k + 2\alpha_k c, c)$.

We answer positively one of the open questions of [17], showing that the time complexity exponent of our method decreases strictly for each k (see Figure 1 for a comparison).

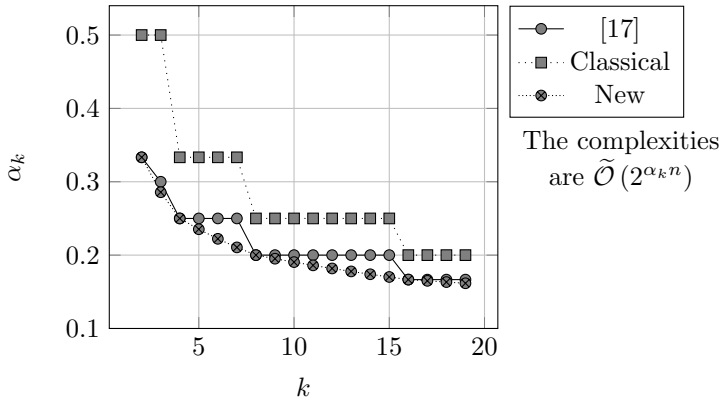

Fig. 1: Comparison of time complexity exponents between the classical case, the algorithms of [17] and our new results, in the qRAM setting.

In the low-qubits setting, we find the following. Except in the cases $k = 3$ and $k = 5$, quantum optimal merging trees give an exponential time speedup for half of the values of k, where the merging is mostly done classically. This also answers a question in [17] (see Figure 2 for a comparison).

Theorem 2. *Let* $k > 2, k \neq 3, 5$ *be an integer and* $\kappa = \lfloor \log_2(k) \rfloor$. *The best quantum merging tree finds a k-xor on n bits in quantum time and classical memory* $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$ *where:*

$$\alpha_k = \begin{cases} \frac{1}{\kappa+1} & \text{if } k < 2^\kappa + 2^{\kappa-1} \\ \frac{2}{2\kappa+3} & \text{if } k \geq 2^\kappa + 2^{\kappa-1} \end{cases}$$

The same method finds 2^{nc} *k-xor with a (quantum time and classical memory) complexity exponent of* $n \max(\alpha_k + \alpha_k c, c)$.

3.2 Quantum Algorithms for Other Variants

For the other problems considered, we give algorithms in the qRAM setting. We can give some generic results, but the precise optimal complexities are obtained with our Mixed Integer Linear program.

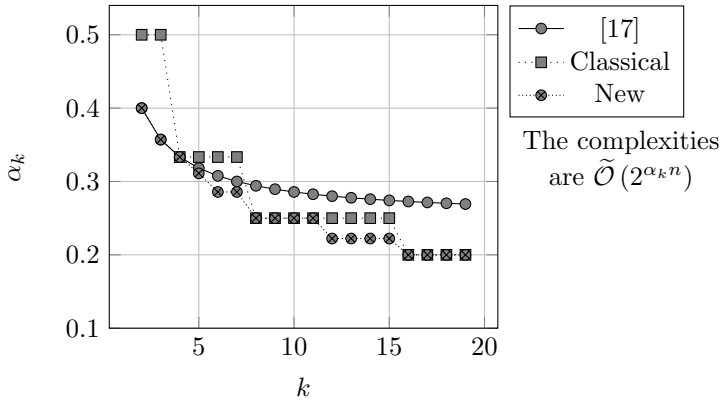

Fig. 2: Comparison of time complexity exponents between the classical case, the algorithms of [17] and our new results, with $\mathcal{O}(n)$ qubits only.

***k*-xor with Classical Lists.** For Problem 1, we prove Proposition 1 and obtain Figure 3.

Proposition 1. *Let $k > 2$ which is not a power of 2, let $\kappa = \lfloor \log_2 k \rfloor$. The quantum time complexity of k-xor* with classical lists *is $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$ with $\alpha_k \leq \frac{1}{2+\lfloor \log_2 k \rfloor}$.*

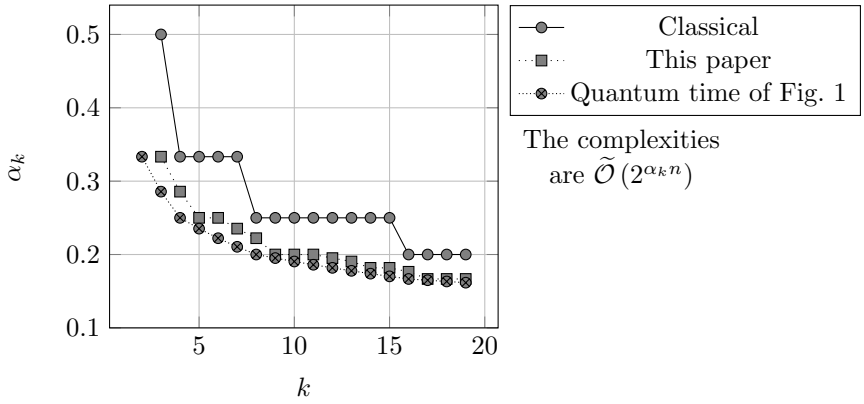

Fig. 3: Quantum time complexities of the k-xor problem with classical input data, compared with the k-xor with a quantum oracle, and with the classical time.

Unique k-xor. For Problems 4 and 3, we give algorithms in the qRAM model starting from $k = 3$. We improve over the previously known techniques for many values of k, as shown on Figure 4. Our time complexity is given by Theorem 3.

Theorem 3. *Let $k > 2$ be an integer. The best merging tree finds, given k lists of uniformly distributed n-bit strings, of size $2^{n/k}$ each, a k-xor on n bits (if it exists) in quantum time $\widetilde{\mathcal{O}}\left(2^{\beta_k n}\right)$ where $\beta_k = \frac{1}{k}\frac{k+\lceil k/5\rceil}{4}$. In particular, it converges towards a minimum 0.3, which is reached by multiples of 5.*

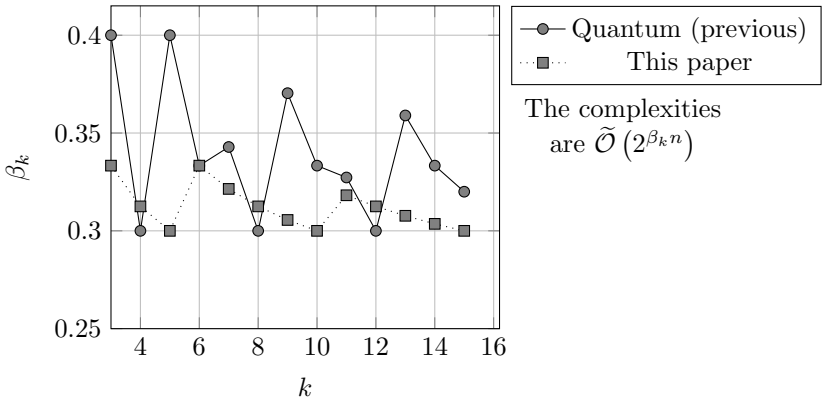

Fig. 4: Unique k-xor time complexity exponent with our method, compared with the previous quantum techniques known ([1] and [5]).

On k-sum Algorithms. We show in Section 4.7 that all our algorithms also apply to the case where modular additions (modulo a power of 2) are considered instead of XORs, with the same complexities.

4 Introducing the k-Merging trees

In this section, we first present Wagner's algorithm [32] in two ways: first, as introduced in [32], second, as an alternative way, which will appear much more compliant with quantum exhaustive search.

Wagner's algorithm is a recursive generalization of an idea introduced by Camion and Patarin [13]. The description in [32] uses lists, but to emphasize the translation to a quantum algorithm, *we will start by considering Problem 2 instead*, with an oracle $H : \{0,1\}^n \to \{0,1\}^n$.

We will next introduce and define the context of k-merging trees. They provide a unified framework for merging quantumly (and classically) and enable automatic search of optimal merging strategies. We will show how to use these trees in the quantum case, and how to optimize them.

4.1 Wagner's Binary Tree in a Breadth-first Order

We now fix the constant k. Wagner notices that given two sorted lists L_1 and L_2 of random n-bit elements, it is easy to "merge" L_1 and L_2 according to some prefix of length u. Let L_u be the lists of pairs $x_1 \in L_1, x_2 \in L_2$ such that $x_1 \oplus x_2$ has its first u bits to zero. We say that such x_1 and x_2 *partially collide* on u bits. Then L_u can be produced in time $\max(|L_u|, \min(|L_1|, |L_2|))$.

For example, if L_1 and L_2 contain 2^u elements and we want the merged list of partial collisions on the first u bits, then this list will have a size of around 2^u and can be obtained in time 2^u.

If k is given, and if H is a random oracle, Wagner's algorithm is a *strategy of successive merges* which builds a sequence of lists of *partial* ℓ-xor on u bits, for increasing values of $u < n$ and $\ell < k$, culminating into a single k-xor. This strategy depends only on n and k.

Example: 4-xor. The strategy for 4-xor is depicted on Figure 5. We start from 4 lists of $2^{n/3}$ random elements each. At the second level of the tree, we build two lists of $2^{n/3}$ partial $\frac{n}{3}$-bit collision (2-xors on $u = n/3$ bits), by merging the two pairs of lists in time $2^{n/3}$. The root is obtained by merging the two lists of collisions, expecting a single result since there are $2^{2n/3}$ 4-tuples to form, with $2n/3$ remaining bits to put to zero.

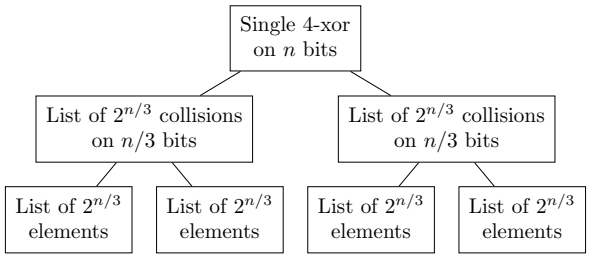

Fig. 5: Structure of Wagner's 4-xor-tree

General k. If k is a power of 2, we write $k = 2^\kappa$. In the remaining of this paper, when k is an integer, we write $\kappa = \lfloor \log_2(k) \rfloor$ for ease of notation. In the context of Wagner's algorithm, if k is not a power of 2, we first take $k - 2^\kappa$ arbitrary elements $z_1, \ldots z_{k-2^\kappa}$ and then find a 2^κ-xor on their sum. So assume without loss of generality that $k = 2^\kappa$. All the lists in the tree will have size $2^{\frac{n}{\kappa+1}}$.

- At the lowest level of the tree (level 0), we build k lists of $2^{\frac{n}{\kappa+1}}$ single elements, making random queries to H.
- At level 1, we merge the lists by pairs, obtaining $2^{\kappa-1}$ lists, each one containing $2^{\frac{n}{\kappa+1}}$ collisions on $\frac{n}{\kappa+1}$ bits.

- At level i ($0 \leq i \leq \kappa - 1$), we have $2^{\kappa-i}$ lists of 2^i-tuples which XOR to zero on $\frac{in}{\kappa+1}$ bits: each level puts $\frac{n}{\kappa+1}$ new bits to zero. Notice that all these bit-positions are arbitrary and fixed, for example prefixes of increasing size.
- At the final level, we merge two lists of $2^{\kappa-1}$-tuples which XOR to zero on $\frac{(\kappa-1)n}{\kappa+1}$ bits, both lists having size $2^{\frac{n}{\kappa+1}}$. We expect on average one 2^κ-tuple to entirely XOR to zero.

4.2 Building a k-tree in a Depth-first Order

The structure of the previous tree defines the merging strategy. But we can build the lists in another order. To build a node of the tree, it suffices to have built its children; not necessarily all nodes of bigger depth. Wagner [32] already remarks that this allows to reduce the memory requirement of his algorithm from 2^κ lists (all the leaves of the tree) to κ.

On Figure 6, we highlight the difference between these two strategies, by considering the 4-xor tree of Figure 5. In a breadth-first manner, we go from one level to the other by building all the nodes (the new nodes are put in bold). Four lists need to be stored (the whole lower level). In a depth-first manner, only two lists need to be stored.

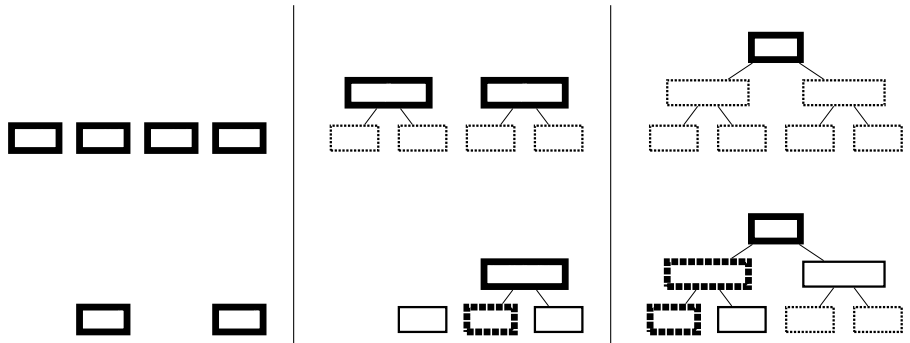

Fig. 6: Building the 4-xor tree of Figure 5 in a breadth-first (above) or depth-first manner (below). At each new step, new lists are built (in bold). We put in dotted the lists which are either discarded at this step, or do not need to be stored.

Example: 4-xor. We illustrate this depth-first tree traversal with the 4-xor example of before. Lists are numbered as in Figure 7.

1. We build and store the list L_0 of $2^{n/3}$ elements.
2. We build the list L_1 of pairs x, x_0 such that $x_0 \in L_0$, x is a new queried element, and $x \oplus x_0$ is 0 on $n/3$ bits. To build a list of $2^{n/3}$ elements, we need time $2^{n/3}$, as each new x has on average one partial collision ($n/3$-bit condition) with some x_0 in L_0 ($2^{n/3}$ elements).

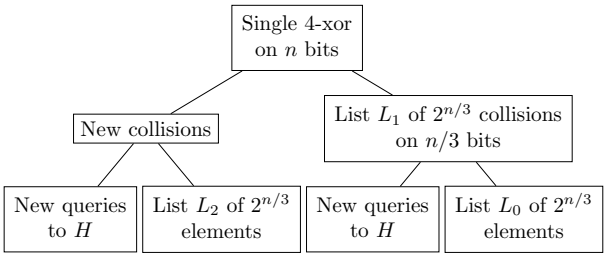

Fig. 7: Depth-first order in which to build the lists.

3. We discard the list L_0. We build and store the list L_2 of $2^{n/3}$ elements.
4. We find a 4-xor on n bits as follows: we make new queries x. Given an element x, we expect a partial $n/3$-bit collision with some $x_2 \in L_2$ (if there is none, abort for this x). Given the value $x \oplus x_2$, we expect a partial $2n/3$-bit collision with some $(x' \oplus x_0) \in L_1$ (if there is none, abort). Then value $x \oplus x_2 \oplus (x' \oplus x_0)$ has $2n/3$ bits to zero. It remains to nullify $n/3$ remaining bits, which is why we repeat this operation for $2^{n/3}$ values of x.

Ensuring a Success Probability of 1. Minder and Sinclair [26] provided a study of the probability of failure in Wagner's algorithm. By building the tree in a depth-first manner, we can easily ensure an exponentially high success probability, that will hold in the quantum setting as well as in the classical. The idea is to always ensure that, given a candidate, a list will yield at least one partially colliding element on the bits that we wish to put to zero. Up to a logarithmic factor in n and constant in k the complexities remain unchanged. The details can be found in Appendix A.2.

4.3 Limitations of the Extension to Quantum k-trees

In [15], Dinur states that "as a first important future research direction, our framework can be adapted to the (increasingly relevant) quantum computation model by combining it with Grover's algorithm."

In the breadth-first variant of Wagner's algorithm, it does not seem easy to use Grover's algorithm as a subroutine, as the initial lists are all fixed. In this case, whenever two lists are merged, the time complexity of this operation is exactly the size of the output list: we cannot expect any quantum improvement if we are to write this list in memory (quantum or classical); and we cannot expect to pursue the tree traversal if we haven't written this list.

This fundamental problem is the main limitation on the quantum k-xor algorithms of [17], in which the authors give two approaches. First, it is possible to mimic Wagner's algorithm. Given a set of queries to H, one reproduces the k-tree and moves from one set to another in the MNRS quantum walk framework [25]. The inherent limitation of this procedure is that it reproduces the classical steps, and cannot yield a better time when k is not a power of 2. Second, they use trees of depth 1: the leaf nodes are produced using some (classical

or quantum) precomputation, and then, they do a Grover search for the final element.

However, in the depth-first variant, each new step corresponds to *some new exhaustive search*, as it begins with the query of new elements x which are *matched* (not merged) against the currently stored lists. Hence, classical search can easily be replaced with quantum search; classical queries to H are replaced with quantum queries. We apply this idea in the next section.

4.4 Examples of Quantum Merging

Let us consider the *depth-first* tree traversal for 4-xor of Figure 7. We now allow quantum computations. Each new node in the tree will be potentially built using *quantum* queries to H and lookups to the previously computed nodes. We reuse the numbering of lists of Figure 7.

1. We build and store the list L_0 of $2^{n/3}$ elements. Quantum computing does not help here.
2. We build the list L_1 of pairs x, x_0 such that $x_0 \in L_0$, x is a new queried element, and $x \oplus x_0$ is 0 on $n/3$ bits. Since the list is of size $2^{n/3}$, and it needs to be written down, Grover search will not accelerate this step. We still need time $2^{n/3}$.
3. We discard the list L_0. We build and store the list L_2 of $2^{n/3}$ elements.
4. To find the final 4-xor, we are testing $2^{n/3}$ values of x, after which we expect that the partial collision with a candidate in L_2 and a candidate in L_1 also nullifies the last $n/3$ bits. This step can be done using Grover search, in time $2^{n/6}$.

At this point, it becomes clear that the tree of Figure 5 must be re-optimized, so that all steps, including the last Grover search, take the same time. This new strategy is specific to the quantum setting; we represent it on Figure 8. We obtain a time complexity of $\widetilde{\mathcal{O}}\left(2^{n/4}\right)$, which is that of [17] for 4-xor. We don't use a quantum walk anymore, but the procedure still requires $\widetilde{\mathcal{O}}\left(2^{n/4}\right)$ qRAM to hold the intermediate lists L_2 and L_1 during the final Grover search.

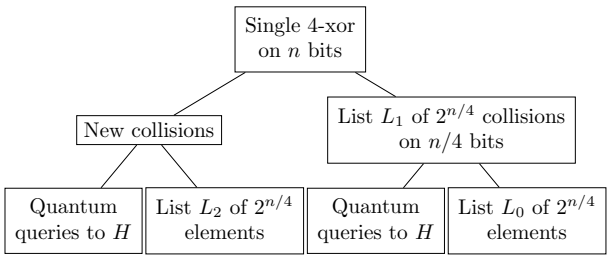

Fig. 8: Quantum 4-xor merging tree.

Classically, the optimality of Wagner's algorithm among all mergings is well-known, and there is no particular incentive to study the depth-first traversal of trees, nor tree structures more general than binary. Quantumly, this is not the case, as examples in [17] have shown. We can see this on the example of 3-xor (Problem 2) in Algorithm 1, which incidentally improves over [17] in the qRAM setting.

Algorithm 1 Quantum 3-xor Algorithm with qRAM

1: Store a list L_0 of $2^{2n/7}$ elements;
2: Using Grover subroutines, build a list L_1 of $2^{n/7}$ elements with a $\frac{2n}{7}$-bit zero prefix;
3: Use Grover's algorithm to find an element x such that $f(x) = 1$, where f is defined as:
- Find $x_0 \in L_0$ which collides with x on the first $\frac{2n}{7}$ bits, in time $\widetilde{\mathcal{O}}(1)$, with probability of success 1 (see Supplementary Material A.2),
- Find $x_1 \in L_1$ such that $x_0 \oplus x_1 \oplus x$ is zero on $\frac{3n}{7}$ bits,
- If $x_0 \oplus x_1 \oplus x = 0$, return 1, else 0.

This requires $\sqrt{2^{4n/7}}$ iterations, as $x_0 \oplus x_1 \oplus x$ has always $\frac{3n}{7}$ bits to zero; there remains $\frac{4n}{7}$ bits to nullify.

This "3-xor-tree" is of depth one, and classically, it does not yield a speedup over the collision exponent $\frac{1}{2}$. The fact that there is an *inherently quantum* merging strategy encourages us to pursue with a general framework.

4.5 Definition of Merging Trees

In order to emphasize that our trees are constructed in a depth-first manner, and to make their definition more suitable, we start from now on to represent them as *unbalanced* trees where each node introduces a new exhaustive search, as on Figure 9.

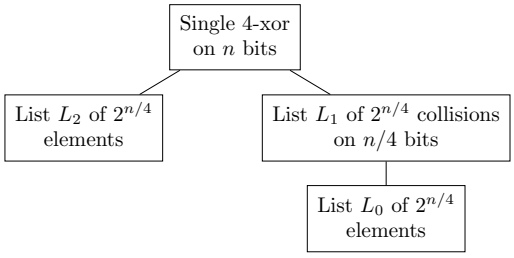

Fig. 9: Tree of Figure 8 as an unbalanced merging tree.

Strategies do not Depend on n. Since all the complexities throughout this paper are exponential in the output bit-size n and we focus on the exponent, we write them in $\log_2$ as $\alpha_k n$ for some α_k which depends only on k. We notice that n is a common factor in all complexities, so it can actually by removed. Next, we define our unbalanced *merging trees*. A tree represents a possible strategy for computing a k-xor; due to our specific writing, its number of nodes is k. Each node corresponds to computing a new list, starting from the leaves, computing the root last.

Definition 1. *A k-merging tree is defined recursively as follows:*

- *If $k = 1$, it has no children: this corresponds to "simple queries" to H.*
- *If $k > 1$, it can have up to $k - 1$ children $T_0, \dots T_{\ell-1}$, which are k_i-merging trees respectively, with the constraint $k_0 + \dots + k_{\ell-1} = k - 1$.*

In other words, a k-sum to zero can be obtained by summing some k_i-sums, such that the k_i sum to k (here a $+1$ comes from the exhaustive search at the root of the tree).

Next, we label each node of the tree with some variables, which represents the characteristics of the list computed. We obtain the general shape of a tree represented on Figure 10.

- The number ℓ of nodes of the subtree
- The number u of bits to zero (as a multiple of n)
- The size s of this list: s represents a size of 2^{sn}
- The (time) cost c of producing this list: c represents a time complexity of 2^{cn}

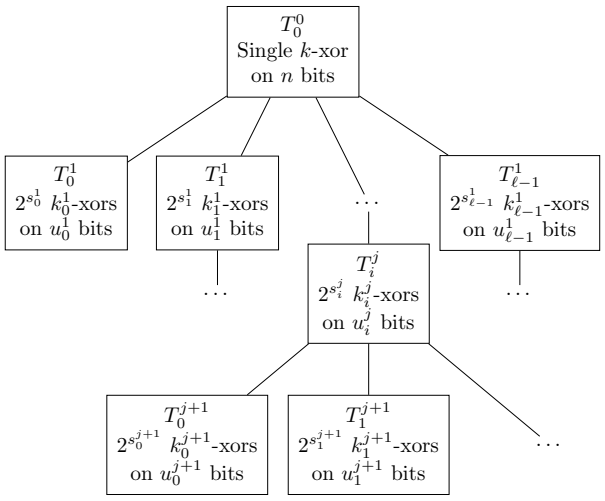

Fig. 10: k-merging tree

The Merging Strategy. We now consider a k-node T and the ℓ subtrees (of children) $T_0, \dots T_{\ell-1}$ attached to it. We suppose that they are ordered by their number of nodes (hence the lists will contain k_0-xors, k_1-xors, ..., $k_{\ell-1}$-xors, with $k_0 + \dots k_{\ell-1} + 1 = k$). The *merging strategy* is inherent to the definition of merging trees, and independent of the computation model. It generalizes the depth-first examples of Section 4.

Each element of T is built using exhaustive search, with $T_0, \dots T_{\ell-1}$ as intermediate data. We impose that the zero-prefixes of $T_0, \dots T_{\ell-1}$ are contained in one another. Let $u_0, u_1, \dots u_{\ell-1}$ be the sizes of these prefixes and $s_0, s_1, \dots s_{\ell-1}$ the sizes of the lists. Given x in the search space of T, the test proceeds in ℓ steps. First, we first make sure that x has zero-prefix u_0. Then we can match it with the first child T_0. Since this child contains 2^{s_0} elements, we can expect to find $x_0 \in T_0$ such that $x \oplus x_0$ has $u_0 + s_0$ bits to zero. Now we search T_1 for some x_1 which increases the number of zeroes in $x \oplus x_0 \oplus x_1$. We would like T_1 to have a zero-prefix of size $u_1 = u_0 + s_0$. Then $x \oplus x_0 \oplus x_1$ will have $u_1 + s_1 = u_0 + s_0 + s_1$ zero, and so on.

We see that for this depth-first merging strategy to work, we need a constraint relating the sizes of the lists and of the prefix of each node. It must hold at any non-leaf node in the tree.

Constraint 1 (A pyramid of zeroes) *Let $T_0, \dots T_{\ell-1}$ be the ℓ subtrees attached to a given k-node T, ordered by their number of nodes. Let $u_0, u_1, \dots u_{\ell-1}$ be their prefix sizes and $s_0, s_1, \dots s_{\ell-1}$ be their sizes. We have:*

$$\forall 1 \leq i \leq \ell - 1, u_i = u_{i-1} + s_{i-1} \ .$$

In other words, given x in the search space for node T, having u_0 zeroes, we expect only one candidate $x_0 \in T_0$ such that $x_0 \oplus x_1$ has u_1 zeroes, one candidate in T_1, *etc.* This constraint also ensures a success probability of 1 by the argument of Section 4.2. Since the list of node T_i is responsible for putting $u_{i+1} - u_i$ bits to zero exactly, we ensure that it takes all the values in this range. Notice that at this point, our definition of merging trees encompasses the binary tree of Wagner's algorithm, created in a depth-first manner.

Computation of the cost of a Tree. Since the goal of our strategy is to obtain the best time complexity for merging, we enforce *computational* constraints, which relate the *cost* of a k-node T with his size and zero-prefix and that of its children. These constraints depend on the computation model used; whether we authorize classical or quantum computation, qRAM or not.

Constraint 2 (Cost of a leaf node) *A leaf node T with size s and zero prefix u has a cost c such that classically $c = u + s$ and quantumly $c = s + \frac{u}{2}$.*

Classically, finding a single x with a prefix of u bits requires 2^u queries to H. Quantumly, it requires $2^{u/2}$ superposition queries with Grover's algorithm.

Constraint 3 (Cost of a non-leaf node) *A k-node T with size s and zero prefix u, with children $T_0, \dots T_{\ell-1}$ having sizes $s_0, \dots s_{\ell-1}$ and prefixes $u_0, \dots u_{\ell-1}$ has a cost c such that:*

- *Classically* $c = s + u + u_0 - u_{\ell-1} - s_{\ell-1}$
- *Quantumly, with qRAM:* $c = s + \frac{1}{2}(u + u_0 - u_{\ell-1} - s_{\ell-1})$
- *Quantumly, without qRAM:* $c = s + \frac{1}{2}(u - u_{\ell-1} - s_{\ell-1}) + \max\left(\frac{u_0}{2}, s_0, \ldots, s_{\ell-1}\right)$

In the classical setting, there are 2^s elements in the node to build and u zeroes to obtain. We must start from an element with u_0 zeroes, which requires already 2^{u_0} queries. Next, we traverse all intermediate lists, which give us a k-xor on $u_{\ell-1} + s_{\ell-1}$ zeroes. There remains $u - u_{\ell-1} - s_{\ell-1}$ zeroes to obtain, so we have to repeat this $2^{u-u_{\ell-1}-s_{\ell-1}}$ times. Quantumly, if we have qRAM access to the previously computed children, we use Grover's algorithm. We take the square root of the classical complexity for finding one element and multiply it by 2^s, the total number of elements in the node. If we don't have qRAM access, we can emulate a qRAM by a sequential lookup of classically stored data. This was done in [14] in the case of quantum collision search (2-xor) and further used in [17] in some k-xor algorithms. Checking whether $x \in T_i$ can be done in time $n2^{s_i}$ using a sequence of comparisons. Finding a partially colliding element on some target takes the same time. Since each child list is queried this way, for each iteration of Grover search, the time complexity becomes:

$$2^{s+\frac{1}{2}(u-u_{\ell-1}-s_{\ell-1})}\left(2^{\frac{u_0}{2}} + 2^{s_0} + \ldots + 2^{s_\ell}\right) \quad .$$

We approximate the right sum by $2^{\max\left(\frac{u_0}{2}, s_0, \ldots, s_\ell\right)}$. This remains valid up to a constant factor in k. In the quantum setting, we will also authorize to fall back on classical computations if there is no better choice.

Finally, the size and number of zeroes of the final list (the root node) are parameters of the problem.

Constraint 4 (Final number of solutions) *The root T of the tree has zero-prefix $u = 1$ (since it requires n zeroes). Its size s is 0 if we want a single tuple, or γ if we want $2^{\gamma n}$ of them for some constant γ.*

Example. We can take as example Algorithm 1, which builds a 3-xor using two intermediate lists. We have a merging tree T, where the root has children T_0 and T_1. At T_0, we build a list of $2^{2n/7}$ elements: $u_0 = 0, s_0 = \frac{2}{7}$. At T_1 we build a list of $2^{n/7}$ elements with a $\frac{2n}{7}$-bit zero prefix: $u_1 = \frac{2}{7}$, $s_1 = \frac{1}{7}$. At the root we have $s = 0$ and $u = 1$. The costs of all nodes are $c_0 = c_1 = c = \frac{2}{7}$. We can verify that $u_1 = u_0 + s_0$ and $c = s + \frac{1}{2}(u + u_0 - u_1 - s_1) = 0 + \frac{1}{2}(1 - 1/7 - 2/7) = \frac{2}{7}$.

4.6 Optimization of Merging Trees

The description of merging trees that we have given above has two purposes: first, to provide a unified framework for merging quantumly and classically; second, to enable automatic search of optimal merging strategies. Given a tree structure, minimizing the total time complexity (the maximum of c_i for all T_i) is a linear problem. Given k, we can try different possible tree structures and find an optimal one.

After finding some punctual improved algorithms for some values of k but not obvious relation or generalization at that point, we decided to use Mixed Integer Linear Programming (MILP) in order to find the best possible merging trees. The results of this linear program allowed us to overcome the *a priori* lack of intuition, to understand the optimal merging trees and to obtain the generalized algorithms that we will present in the next sections.

Linear Program. We want to minimize the total time complexity of the merging tree. By definition of c_i, this is the sum of all 2^{nc_i} for all nodes T_i, starting from the leaf nodes (which are traversed first) up to the root (which is produced last). We approximate it to $2^{n \max_i(c_i)}$, up to a constant factor in k. Hence we minimize $c = \max_i(c_i)$ under the constraints outlined above.

Adaptations. The constraints of Section 4.5 are the only ones required to solve efficiently Problem 2. We will amend the framework in Section 6 to solve efficiently Problems 1, 4 and 3.

4.7 Extension to Modular Additions

We elaborate on the case where the bitwise XOR operation is replaced by modular additions. First, we recall that the classical k-tree algorithm [32] is already adapted to the k-sum case. Suppose that all the elements lie in the interval $[-M; M]$ where M is some modulus, and we are looking for a k-sum to zero. Instead of using lists of partially colliding elements, we look for sums falling in successive reduced intervals $\left[-\frac{M}{2^\ell}; \frac{M}{2^\ell}\right]$. The computing cost is exactly the same as before.

If we replace XOR by additions modulo 2^n, we can easily show that the merging tree complexities are not impacted. Consider a node T with children $T_0, \dots T_{\ell-1}$. With XORs, in T's exhaustive search procedure, we take an element x and find successive candidates $x_0, \dots x_{\ell-1}$ increasing the number of zeroes of the sum. With modular additions, we can do the same if we keep a carry bit alongside the partial collisions. More precisely: we find x_0 such that $x + x_0$ has its u_0 less significant bits to zero. Then we find x_1 such that $x_1 + x_0 + x$ has its u_1 less significant bits to zero. We only need to propagate a carry bit. The merging strategy is unchanged, so the time and memory complexities are unchanged.

5 Optimal Merging Trees

In this section, we present our main results regarding Problem 2. We first describe the shape of the optimal trees, and next, the complexities in the qRAM and in the low-qubit setting. Our results are compared with the ones from [17] on Figures 1, 2 and Table 1 in Appendix. Then explain the complexities, which are optimal among all merging strategies represented by the trees of Section 4

5.1 Description of the Optimal Trees

By testing the different possible merging trees, and optimizing each tree with a MILP solver, we obtained optimal merging-tree strategies for solving the k-xor problem in the quantum setting, improving on [17] for many values of k. In the qRAM case, we verify that the quantum walk of [17] is subsumed by the merging tree method for all powers of 2. For non-powers of 2, we reach new and strictly better complexity exponents for all k. In the low-qubits case, we obtain non-trivial improvements for $k = 5, 6, 7$ and a new quantum speedup for half the values of k.

Optimal Trees. First of all, we define a family of trees $\mathcal{T}_k$ which will represent some optimal strategies for k-xor. The root of $\mathcal{T}_k$ (a k-xor) has $\lceil \log_2(k) \rceil$ children. The first child contains $\lfloor \frac{k}{2} \rfloor$-xors on some bits, the second contains $\left\lfloor \frac{1}{2} \left(k - \left\lfloor \frac{k}{2} \right\rfloor\right) \right\rfloor$-xors. In general, child i contains k_i-xors, and child $i+1$ contains $k_{i+1} = \left\lfloor \frac{1}{2} \left(k - \sum_{j=1}^{i} k_j\right) \right\rfloor$. The children subtrees are all $\mathcal{T}_{k_i}$.

If the $\mathcal{T}_k$ trees are solved with the classical constraints, we recover the complexities of Wagner's algorithm. Quantumly, we can make use of the additional nodes when k is not a power of 2. Indeed, Grover's algorithm allows to create elements with some zero-prefix quadratically faster. This is the source of the 3-xor quantum speedup (see Algorithm 1), and it can be generalized. We point out that $\mathcal{T}_k$ provides the optimal complexity both in the qRAM and low-qubits setting (for $k > 5$) however it is not the only merging tree with such optimization.

qRAM Case. In the qRAM case, each node that has a non-empty zero prefix is produced using Grover search. We note $\kappa = \lfloor \log_2(k) \rfloor$ and $\alpha_k = 2^\kappa \frac{1}{(1+\kappa)2^\kappa + k}$. In the optimization of $\mathcal{T}_k$, all the nodes have exactly the same cost (so all the lists are generated in the same quantum time). For all nodes of the tree, the optimal values of s_i and u_i are multiples of $\frac{1}{(1+\kappa)2^\kappa + k}$. The whole description of the optimal tree is easily derived from the constraints, but we do not have a clear description of it for a given k. We give the tree and constraints in the example of 11-xor in Appendix B.

Low-qubits Case. In the low-qubits case, for $k \neq 2, 3, 5$, the best strategy is always to use classical searches, except at some leaves of the tree, where some elements with zero-prefixes are produced using Grover search. This gives one intermediate level of complexity between two successive powers of 2. For collision search, we obtain the algorithm of [14] with $\alpha_2 = \frac{2}{5}$. For $k = 3$, we obtain the algorithm of [17] with $\alpha_3 = \frac{5}{14}$, showing that it remains optimal in our extended framework (contrary to 3-xor with qRAM, see Algorithm 1). The case $k = 5$ is the last using Grover search at the root of the tree, with a surprisingly non-trivial $\alpha_5 = \frac{14}{45}$. We describe it in full detail in Appendix B.

Memory. The memory of our algorithms, for an equal time, is always equal or better than the one from [17], in both settings. Notice that the low-qubits variants actually use classical memory only (it can be seen as a quantum-classical tradeoff), its $\mathcal{O}(n)$ qubits being dedicated to computing. For a time $\widetilde{\mathcal{O}}(2^{\alpha_k n})$, the qRAM variant requires $\widetilde{\mathcal{O}}(2^{\alpha_k n})$ qRAM (it is needed to store the leaf lists).

5.2 Optimality in the qRAM Setting

The MILP experiments helped us find the time complexity exponents α_k for $k \leq 20$, and acquire an intuition of the optimal algorithms for any k. We can prove this optimality in the qRAM setting among all merging trees.

Theorem 1. *Let $k \geq 2$ be an integer and $\kappa = \lfloor \log_2(k) \rfloor$. The best quantum merging tree finds a k-xor on n bits in quantum time (and memory) $\widetilde{\mathcal{O}}(2^{\alpha_k n})$ where $\alpha_k = \frac{2^\kappa}{(1+\kappa)2^\kappa + k}$. The same method finds 2^{nc} k-xor with a quantum (time and memory) complexity exponent of $n \max(\alpha_k + 2\alpha_k c, c)$.*

Furthermore, for every k, the optimum is realized by $\mathcal{T}_k$.

One can verify that α_k gives the expected exponent for powers of 2, where it is equal to $\frac{1}{\kappa+2}$.

The idea of the proof is an induction on k. It is possible to prove that, if the last child of the root node is a list of partial k_ℓ-xors, then the optimal exponent α_k satisfies:

$$\frac{1}{\alpha_k} \leq 1 + \frac{1}{2\alpha_{k-k_\ell}} + \frac{1}{2\alpha_{k_\ell}} \ .$$

This is where the structure $\mathcal{T}_k$ appears naturally. Since α_k is a decreasing function of k, to minimize the sum on the right, we need k_ℓ equal to $\lfloor k/2 \rfloor$. By plugging in this value and using the recurrence hypothesis, we obtain immediately the formula for α_k, and show that it is attained by $\mathcal{T}_k$. The full proof is given in Appendix B.3.

5.3 Theoretical Result in the low-qubits Setting

In the low-qubits setting, we can explain why Theorem 2 gives the optimal complexities.

Theorem 2. *Let $k > 2, k \neq 3, 5$ be an integer and $\kappa = \lfloor \log_2(k) \rfloor$. The best quantum merging tree finds a k-xor on n bits in quantum time and classical memory $\widetilde{\mathcal{O}}(2^{\alpha_k n})$ where:*

$$\alpha_k = \begin{cases} \frac{1}{\kappa+1} & \text{if } k < 2^\kappa + 2^{\kappa-1} \\ \frac{2}{2\kappa+3} & \text{if } k \geq 2^\kappa + 2^{\kappa-1} \end{cases}$$

The same method finds 2^{nc} k-xors with a (quantum time and classical memory) complexity exponent of $\max(\alpha_k n + \alpha_k c, c)$.

Furthermore, for every $k \neq 3, 5$, the optimum is realized by $\mathcal{T}_k$.

Informally, when k is bigger than 6, the merging operation at the root of the tree is performed using classical search. Grover search cannot be used anymore, as each iteration requires to pay the full length of the children (to emulate the qRAM lookups). In that case, we single out the first child T_0. We can rewrite the k-tree as a single merge between T_0, which is a k_0-tree, and a $k - k_0$-tree. The costs of producing these trees should be balanced, hence we should have $k_0 = \lfloor k/2 \rfloor$ as before, and we obtain the tree $\mathcal{T}_k$. Now we can remark that if $k < 2^\kappa + 2^{\kappa-1}$, then $\lfloor k/2 \rfloor < 2^{\kappa-1} + 2^{\kappa-2}$; and conversely, if $k \geq 2^\kappa + 2^{\kappa-1}$, then $\lfloor k/2 \rfloor \geq 2^{\kappa-1} + 2^{\kappa-2}$. In other words, we fall back very easily on the recurrence hypothesis.

6 Domain Restriction and the k-List Problem

In this section, we extend merging trees to a much broader setting, which enables us to provide, for the first time, quantum algorithms for solving Problem 1, where quantum oracle calls are replaced by classical data. We also show how we can limit the input domain size, solving Problems 3 and 4 with time complexities better than the previous algorithms for most of the values of k.

First we will present an overview of related work, then how to adapt the merging trees of Section 4 to this new situation. We will present some examples of algorithms and our general results. Recall that in our formulation of Problems 4 and 3, the *input domain* of the oracle H is restricted to n/k bits and the codomain is n bits; alternatively, the input lists are of size smaller than $2^{n/k}$.

6.1 Overview of Previous Related Work

Extended k-tree Algorithm of [26]. Minder and Sinclair limit the sizes of the lists at the first level of Wagner's k-tree. This corresponds to taking an oracle $H : \{0,1\}^{dn} \to \{0,1\}^n$ with $d < 1$. The authors use MILP to derive the optimal list sizes depending on the domain restriction. Their optimal algorithms roughly run in two steps: in the first levels of the binary tree, all pairs of elements are produced, increasing the list sizes; after that, classical merging is used.

Dissection Algorithms. In [16], the authors study a family of *bicomposite* problems with a single solution, which include hard knapsacks and multiple-encryption. They generalize the technique of Schroeppel and Shamir [31] to improve the memory complexity of these problems. Their method consists in guessing some intermediate values, then producing efficiently lists of partial guesses, before matching them. A bigger meet-in-the-middle instance is broken down into smaller ones.

Quantum Algorithms for Multiple Encryption. In [24], Kaplan proves that 2-encryption is (quantumly) equivalent to element distinctness, solved by an optimal algorithm of Ambainis [1] based on a quantum walk, in time and memory $\widetilde{\mathcal{O}}\left(2^{n/3}\right)$ instead of $2^{n/2}$ classically. The quantum walk of Ambainis' algorithm,

and its qRAM usage, seems the quantum counterpart of the classical meet-in-the-middle approach, and decreasing the memory used seems unlikely. However, k-encryption for higher values of k looks promising. A quantum walk (originally designed for solving subset sums) is given in [5], solving Problem 3 for $k = 4$ in time $\widetilde{\mathcal{O}}\left(2^{0.3n}\right)$. This represents an exponential quantum time improvement with respect to the element distinctness problem (or single-solution collision search). However, this algorithm cannot be used for 4-encryption. Indeed, in the quantum optimization, the size of the "intermediate value" that is guessed is not a multiple of $n/4$ bits. This has no consequence on Problem 3, but if we try to translate the algorithm to attack multiple-encryption, we cannot solve efficiently the smaller meet-in-the middle problems [b]. Natural questions are: whether there exists also such an improvement for 4-encryption [c], whether other values of k than a multiple of 4 can reach the exponent 0.3 and whether it can be improved.

6.2 Generalized Merging Trees for Problems 1, 3 and 4.

Our observation is that the *dissection* technique of [16, Section 3] finds a very simple analogue in terms of merging trees.

We remark that a merging tree as defined in Section 4 has many unused degrees of freedom. Indeed, suppose that we are building a tree T with children $T_0, \dots T_{\ell-1}$. Each T_i has a zero-prefix of u_i bits. We deliberately used the term "zero-prefix", but we can actually take any value for these bits. During a search for a new element of T, we still look for successive collisions, but the values required depend on the prefixes of each child. All the prefixes are ours to choose, except for the root, since we still want the final k-tuple to XOR to zero.

This allows to *repeat* the node T up to $2^{u_0} \times 2^{u_1} \times \ldots \times 2^{u_{\ell-1}}$ times, and to overcome a limitation in the domain size. We write a merging tree as before, but expect only a small probability of success for the search at the root; so we interleave this tree with repetitions. The root search can be performed many more times, by changing the children.

The final time complexity depends on the complexity of the children, and the number of times that they are repeated. Indeed, suppose that the children $T_0, \dots T_{\ell-1}$ are built in time $t_0, \dots t_{\ell-1}$ (all of this in $\log_2$ and multiples of n). Suppose also that the root search requires time t. With a total number of repetitions r before we find a solution, the children will respectively be repeated $r_0, \dots r_{\ell-1}$ times (up to the choices they have in their prefixes) with $r_0 + \ldots + r_{\ell-1} = r$. We can write the time complexity as:

$$r_0(t_0 + r_1(t_1 + \ldots) \ldots + t)$$

by taking an arbitrary order for the children and writing the algorithm as ℓ nested loops:

[b] Intuitively, it would require to produce efficiently (in time $2^{0.8n}$), from $2^{0.8n}$ choices of k_1 and k_2, the list of $2^{0.8n}$ pairs k_1, k_2 such that $E_{k_1} \circ E_{k_2}(P)$ has some fixed $0.8n$-bit prefix.

[c] Kaplan [24] also gives an algorithm for 4-encryption, but we are in discussion with the author for figuring out a possible error in the complexity. So we will only consider [5].

0. The first loop iterates r_0 times on child T_0

1. Inside the first loop, after building T_0, the second loop iterates r_1 times on child T_1

...

$\ell - 1$ Inside all $\ell - 1$ previous loops, after building $T_0, \dots T_{\ell-2}$, the ℓ-th loop iterates on child $T_{\ell-1}$. Inside this loop:

- We build the child T_ℓ
- We perform the exhaustive search of the root T, using the children $T_0, \dots T_{\ell-1}$

In particular, this method subsumes the algorithms of [16, Section 3] in a classical setting. It also generalizes the idea of guessing intermediate values (which are the prefixes of the children T_i) and running an exhaustive search of these, and extends [16, Section 3] to all intermediate domain sizes.

The quantum correspondence works in a very simple way, like in [7]: these ℓ nested loops become ℓ nested Grover searches. We do not search among the lists T_i themselves, but among choices for T_i, *i.e.* choices for the fixed prefix. The setup (producing the superposition over the whole search space) remains easy. The test of a choice performs the nested computations: creating the list T_i itself and running the other searches.

Example: Quantum and Classical 4-dissection. We take the example of Problem 3. We suppose quantum access to a random function $H : \{0,1\}^{n/4} \to \{0,1\}^n$. Classically, the best algorithm is Algorithm 2, from [31], in time $2^{n/2}$ and memory $2^{n/4}$. Quantumly, the best algorithm is in [5], in time $2^{0.3n}$ using $2^{0.2n}$ qRAM. Our method is Algorithm 3. It runs in quantum time $2^{0.3125n}$, smaller than a simple meet-in-the-middle, and qRAM $2^{0.25n}$. It is worse than [5] for Problems 4 and 3, but it can be used to attack 4-encryption.

Algorithm 2 Classical 4-dissection

1: Query H and store all the elements $H(x)$ in a list L_0
2: **for** each $u \in \{0,1\}^{0.25n}$ **do**
3: Create the list L_1 of pairs x, y with $x \oplus y = u|*$. This takes time $2^{0.25n}$, L_1 contains $2^{0.25n}$ elements (indeed, for each element $x \in L_0$ we expect a partial collision on $0.25n$ bits with some other element $y \in L_0$).
4: **for** each $z \in L_0$ **do**
5: Find $t \in L_0$ such that $t \oplus z = u|*$.
6: Find $x \oplus y \in L_1$ such that $x \oplus y \oplus z \oplus t$ gives a $0.5n$-bit zero prefix.
7: If $x \oplus y \oplus z \oplus t$ is all-zero, then return this result.
8: **end for**
9: **end for**
10: Return the 4-tuple that XORs to zero.

Algorithm 3 Optimal merging tree algorithm for Problems 4 and 3 with $k = 4$

Query H and store all the elements $H(x)$ in a list L_0
for each $u \in \{0,1\}^{0.25n}$ **do**
  for $2^{0.125n}$ repetitions **do**
    Build a list L_1 of $2^{0.125n}$ partial collisions $x \oplus y = u|*$, in time $2^{0.125n}$, using exhaustive search with L_0 as intermediate (if we take any element, we expect a partial collision on $0.125n$ bits with some other in L_0)
    for each $z \in L_0$ **do**
      Find $t \in L_0$ such that $z \oplus t = u|*$
      Find $x \oplus y \in L_1$ that collides with $z \oplus t$ on $0.25n$ more bits
      If $x \oplus y \oplus z \oplus t = 0$, then return this result
    end for
  end for
end for
Return the 4-tuple that XORs to zero.

The classical time complexity of Algorithm 3 would be:

$$\underbrace{2^{0.25n}}_{\text{choice of } u} \left(2^{0.125n} \left(\underbrace{2^{0.125n}}_{\substack{\text{Intermediate} \\ \text{list } L_1}} + \underbrace{2^{0.25n}}_{\substack{\text{Exhaustive} \\ \text{search}}} \right) \right) = 2^{0.625n}$$

which is not optimal. However, as a quantum algorithm with nested Grover searches, it optimizes differently, since exhaustive search factors are replaced by their square roots:

$$2^{0.125n/2} \times 2^{0.125n/2} \left(2^{0.125n} + 2^{0.25n/2} \right) = 2^{0.3125n} \quad .$$

6.3 Quantum Algorithms for Unique k-xor

In what follows, we solve together Problem 4 and 3 in the qRAM model, with the same time complexities. Indeed, if the input data is classical, but if we can use qRAM, we can simulate quantum access to the data by qRAM queries: accessing data in superposition is precisely what the qRAM model allows. We also get the same memory complexities for $k \geq 4$, since the cost of storing the whole domain of size $2^{n/k}$ is not dominant anymore. For $k = 3$, there is a difference in the memory complexity. For completeness, the two procedures for $k = 3$ are given in Appendix C.1.

In the general case, we obtain Figure 4 and Theorem 3. We do not improve the complexity of $2^{0.3n}$ for unique 4-xor obtained in [5], but we converge towards it, we reach it when k is a multiple of 5 and improve on previous works when k is not a multiple of 4. Furthermore, our algorithms can be naturally converted to k-encryption: the prefixes that we guess are always of size given in multiples of n/k bits, so we remain in a similar situation as [16].

From our observations, we derive the optimal merging-tree time complexity for Problems 4 and 3. When k is a multiple of 5, we can just apply our 5-xor algorithm with an increased domain size, and obtain an exponent 0.3. For other values of k, a good combination of Grover searches allows to approach it. While this seems easy to infer, optimizing the quantum memory complexity of this method would require more work.

Theorem 3. *Let $k > 2$ be an integer. The best merging tree finds, given k lists of uniformly distributed n-bit strings, of size $2^{n/k}$ each, a k-xor on n bits if it exists in quantum time $\widetilde{\mathcal{O}}\left(2^{\beta_k n}\right)$ where $\beta_k = \frac{1}{k}\frac{k+\lceil k/5\rceil}{4}$. In particular, it converges towards a minimum 0.3, which is reached by multiples of 5.*

6.4 Quantum Algorithms for Problem 1

We now go back to the k-xor problem with "unrestricted domain", but we remove quantum oracle access and authorize classical inputs only. In that case, qRAM access seems necessary to achieve a time speedup against Wagner's algorithm, so as before, we present results only in the qRAM case.

The quantum algorithms based on merging trees are a succession of Grover searches. The main issue is that, contrary to classical k-xor, where the domain size is exactly given by the leaf lists in the merging tree, the Grover searches can span a bigger search space than the leaf list sizes. If the input lists are given classically, it is still possible to query them in superposition, thanks to qRAM. We can also assume that they are sorted (this is a plus). However, the search space in the Grover procedure is now limited by the total number of initial elements.

A solution to this problem is to repeat the subtrees, as we did for the unique k-xor problem. This does not cost many more elements, since obtaining a zero-prefix or an arbitrary-prefix can be done by reusing the same search space. We can also understand that, since the algorithms of [17] are particular cases of merging trees, without the extension with repetitions, they could not be applied for the k-xor with classical inputs. As an example, we give a 4-xor algorithm (Algorithm 4) of quantum time complexity: $2^{n/7}\left(2^{n/7}+2^{n/7}\right) = 2^{2n/7}$.

We can remark the following simple property when k is not a power of 2. For most values of k (including powers of 2), the exponent is actually better, but it seems difficult to find a clear pattern for these improvements or to prove some optimality result.

Proposition 1. *Let $k > 2$ which is not a power of 2, let $\kappa = \lfloor \log_2 k \rfloor$. The quantum time complexity of k-xor* with classical lists *is $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$ with $\alpha_k \leq \frac{1}{2+\lfloor \log_2 k \rfloor}$.*

Proof. We give a general algorithm for k-xor without quantum oracle access, when k is not a power of 2. Suppose that $k = 1 + 2^\kappa$ (the other degrees of freedom will be simply dismissed). We consider the quantum merging tree for 2^κ. It is completely classical, except the last Grover search, for the root. In

Algorithm 4 Quantum algorithm for the 4-list problem with qRAM

Store the input lists L_0, L_1, L_2, L_4 of size $2^{2n/7}$ of n-bit elements in qRAM
for $u \in \{0,1\}^{2n/7}$ **do**
  Build a list L of $2^{n/7}$ pairs $x, y \in L_0 \times L_1$ with $x \oplus y = u|*$, by taking random elements x in L_0 and looking for a $y \in L_1$ which has this $2n/7$-bit relation
  for $z \in L_2$ **do**
    Find $t \in L_3$ such that $z \oplus t = u|*$
    Find, if it exists, $x \oplus y \in L$ such that $x \oplus y \oplus z \oplus t = 0$
  end for
end for

particular, the last search must nullify $\frac{2}{2+\lfloor \log_2 k \rfloor} n$ bits, which is why it runs in quantum time $2^{\frac{n}{2+\lfloor \log_2 k \rfloor}}$.

To build all the nodes except the root, we only need $2^{\frac{1}{2+\lfloor \log_2 k \rfloor} n}$ classical inputs. However, the root itself would require a search space of size $2^{\frac{2}{2+\lfloor \log_2 k \rfloor} n}$. To overcome this limitation, we use the new degree of freedom that we just added. We perform a Grover search on *pairs of elements*. □

7 Applications

7.1 Attacks on Modes of Operations

In [27, Section 3.2], a forgery attack on the COPA mode is shown, which exactly requires to find a 3-xor among sets of encryption and decryption queries made to the mode. Classically, the query complexity decreases from $2^{n/2}$ (standard birthday security) to $2^{n/3}$, breaking initial security claims, but the time complexity does not have such improvement.

Quantumly, there are more consequences. First, if superposition queries are allowed to the mode, the time complexity for finding a 3-xor (in the qRAM model) decreases to $2^{2n/7} = 2^{0.286n}$ instead of the quantum birthday bound $2^{n/3}$. This is an improvement with respect to [17].

With our results, we show that a quantum time speedup is possible even when superposition queries are not allowed, so only classical encryption / decryption queries are required. In $2^{n/3}$ quantum time (albeit $2^{n/3}$ qRAM also), one can find a 3-xor between three lists of $2^{n/3}$ classical queries. On the contrary, even with quantum computations, it is impossible to find a collision in less than $2^{n/2}$ classical queries (since such a collision would not exist).

7.2 Approximate k-list Problem

In [8], Both and May introduce and study the *approximate k-list problem*. It can be seen as a generalization of k-xor in which the final n-bit value only needs to

have a Hamming weight lower than αn for some fraction $0 \leq \alpha \leq \frac{n}{2}$ (so the k-xor is the special case $\alpha = 0$).

We remark that the method of [8, Section 3] can be easily adapted to a quantum algorithm. It consists in running a simple k-xor algorithm with a restricted number of bits to put to zero, and to tailor the length of the final list so that it will contain one element of low Hamming weight with certainty. Rewriting the tree in a depth-first manner and using our quantum merging technique, the final list actually never needs to be built. We simply produce its elements and search (with Grover's algorithm) one with small Hamming weight.

7.3 Quantum time – memory Tradeoff for the Subset-sum Problem

Our extended merging tree algorithms for Problem 3 find an application for the subset-sum problem. Namely, we are able to reach a better quantum time – memory product with respect to the current literature for low-density knapsacks. This is not surprising, considering that this was also the case in [16].

Let $a_1, \ldots a_n, t$ be randomly chosen integers on ℓ bits. We are looking for a subset of indices $I \subset \{1, \ldots n\}$ such that $\sum_{i \in I} a_i \equiv t \mod 2^\ell$. The hardness of this problem is related to the density n/ℓ. When $\ell = poly(n)$, and we expect a single solution with high probability, the best classical algorithm [4], runs in time and memory $\widetilde{\mathcal{O}}\left(2^{0.291n}\right)$. The current best quantum algorithm [21] takes time and memory $\widetilde{\mathcal{O}}\left(2^{0.226n}\right)$.

A subset-sum problem can easily be translated to a k-xor problem with a single solution, or a k-sum problem if modular additions are used instead of bitwise. Indeed, it suffices to separate the set $\{1, \ldots n\}$ into k disjoint parts $J_1 \cup \ldots \cup J_k$ and to start from the lists $L_1, \ldots, L_k$, with list L_j containing all the sums $\sum_{i \in I} a_i$ for $I \subset J_j$.

Both the quantum time and memory complexities of the k-xor (or k-sum) problem with a single solution vary with k. Optimizing the time-memory product (more details are given in Appendix C.1), we find that $k = 12$ seems the most interesting, with a product of $\widetilde{\mathcal{O}}\left(2^{5n/12}\right) = \widetilde{\mathcal{O}}\left(2^{0.412n}\right)$ which is less than the previous $0.452n$.

8 Conclusion

Better Quantum k-xor Algorithms. In this paper, we proposed new algorithms improving the complexities from [17] for most values of k in both the qRAM and low-qubits settings We also gave quantum algorithms for the k-xor problem with classical inputs, and when the input size is limited. This enabled us to gave algorithms for k-encryption running exponentially faster than double-encryption, and to reach the best quantum time – memory product known for solving the subset-sum problem. All our algorithms can be used by replacing xors by sums modulo 2^n.

Optimal Strategies from MILP. We defined the framework of *merging trees*, which allows to write strategies for solving k-list problems (classically and quantumly) in an abstract and systematic way. Our optimization results were obtained using Mixed Integer Linear Programming. We used this experimental evidence to move on to actual proofs and systematic descriptions of our optimums.

Future Work. It seems to us likely that the merging trees we defined could be extended with more advanced techniques, inspired by the classical literature on k-list problems. We tried some of these techniques and could not find a quantum advantage so far. However, it may also be of interest to write a single program encompassing all the known classical techniques for better time-memory tradeoffs.

Open Questions. There are many algorithms for subset-sum [4], decoding random linear codes [9], solving the SIS problem [3] that have a k-list flavor; we wonder whether our techniques can be leveraged to improve the current quantum algorithms for these problems. Finally, we have proven some optimality results *among all merging trees*, which is a set of strategies that we carefully defined, but we do not know whether an extended framework could be suitable to improve the quantum algorithms. In particular, the time complexity of our merging tree algorithms for k-encryption encounters a limit $2^{0.3n}$. Whether an extended framework could allow to break this bound remains unknown to us.

Acknowledgements

The authors would like to thank Xavier Bonnetain, André Chailloux, Lorenzo Grassi, Marc Kaplan and Yu Sasaki for helpful discussions and comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 714294 - acronym QUASYModo).

References

1. Ambainis, A.: Quantum Walk Algorithm for Element Distinctness. SIAM J. Comput. 37(1), 210–239 (2007)
2. Amy, M., Matteo, O.D., Gheorghiu, V., Mosca, M., Parent, A., Schanck, J.M.: Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3. IACR Cryptology ePrint Archive 2016, 992 (2016)
3. Bai, S., Galbraith, S.D., Li, L., Sheffield, D.: Improved combinatorial algorithms for the inhomogeneous short integer solution problem. J. Cryptology 32(1), 35–83 (2019), https://doi.org/10.1007/s00145-018-9304-1
4. Becker, A., Coron, J., Joux, A.: Improved generic algorithms for hard knapsacks. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 6632, pp. 364–385. Springer (2011)

5. Bernstein, D.J., Jeffery, S., Lange, T., Meurer, A.: Quantum algorithms for the subset-sum problem. In: PQCrypto. Lecture Notes in Computer Science, vol. 7932, pp. 16–33. Springer (2013)
6. Bernstein, D.J., Lange, T., Niederhagen, R., Peters, C., Schwabe, P.: FSBday. In: Progress in Cryptology - INDOCRYPT 2009. LNCS, vol. 5922, pp. 18–38. Springer (2009)
7. Bonnetain, X., Naya-Plasencia, M., Schrottenloher, A.: Quantum security analysis of AES. IACR Cryptology ePrint Archive 2019, 272 (2019), https://eprint.iacr.org/2019/272
8. Both, L., May, A.: The approximate k-list problem. IACR Trans. Symmetric Cryptol. 2017(1), 380–397 (2017), https://doi.org/10.13154/tosc.v2017.i1.380-397
9. Both, L., May, A.: Decoding linear codes with high error rate and its impact for LPN security. In: PQCrypto. Lecture Notes in Computer Science, vol. 10786, pp. 25–46. Springer (2018)
10. Brassard, G., Hoyer, P., Mosca, M., Tapp, A.: Quantum amplitude amplification and estimation. Contemporary Mathematics 305, 53–74 (2002)
11. Brassard, G., Høyer, P., Tapp, A.: Quantum cryptanalysis of hash and claw-free functions. In: LATIN. LNCS, vol. 1380, pp. 163–169. Springer (1998)
12. Brassard, G., Høyer, P., Tapp, A.: Quantum algorithm for the collision problem. In: Encyclopedia of Algorithms, pp. 1662–1664 (2016)
13. Camion, P., Patarin, J.: The knapsack hash function proposed at crypto'89 can be broken. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 547, pp. 39–53. Springer (1991)
14. Chailloux, A., Naya-Plasencia, M., Schrottenloher, A.: An Efficient Quantum Collision Search Algorithm and Implications on Symmetric Cryptography. In: Advances in Cryptology - ASIACRYPT 2017. LNCS, vol. 10625, pp. 211–240. Springer (2017)
15. Dinur, I.: An algorithmic framework for the generalized birthday problem. Cryptology ePrint Archive, Report 2018/575 (2018), https://eprint.iacr.org/2018/575
16. Dinur, I., Dunkelman, O., Keller, N., Shamir, A.: Efficient dissection of composite problems, with applications to cryptanalysis, knapsacks, and combinatorial search problems. In: CRYPTO. Lecture Notes in Computer Science, vol. 7417, pp. 719–740. Springer (2012)
17. Grassi, L., Naya-Plasencia, M., Schrottenloher, A.: Quantum algorithms for the k -xor problem. In: ASIACRYPT (1). Lecture Notes in Computer Science, vol. 11272, pp. 527–559. Springer (2018)
18. Grassl, M., Langenberg, B., Roetteler, M., Steinwandt, R.: Applying Grover's Algorithm to AES: Quantum Resource Estimates. In: PQCrypto. Lecture Notes in Computer Science, vol. 9606, pp. 29–43. Springer (2016)
19. Grover, L.K.: A Fast Quantum Mechanical Algorithm for Database Search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing 1996. pp. 212–219. ACM (1996), http://doi.acm.org/10.1145/237814.237866
20. Grover, L.K., Rudolph, T.: How significant are the known collision and element distinctness quantum algorithms? Quantum Information & Computation 4(3), 201–206 (2004), http://portal.acm.org/citation.cfm?id=2011622
21. Helm, A., May, A.: Subset sum quantumly in 1.17^n. In: TQC. LIPIcs, vol. 111, pp. 5:1–5:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
22. Hosoyamada, A., Sasaki, Y., Xagawa, K.: Quantum multicollision-finding algorithm. In: ASIACRYPT (2). Lecture Notes in Computer Science, vol. 10625, pp. 179–210. Springer (2017)

23. Jaques, S., Schanck, J.M.: Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. IACR Cryptology ePrint Archive 2019, 103 (2019)
24. Kaplan, M.: Quantum attacks against iterated block ciphers. CoRR abs/1410.1434 (2014), http://arxiv.org/abs/1410.1434
25. Magniez, F., Nayak, A., Roland, J., Santha, M.: Search via Quantum Walk. SIAM J. Comput. 40(1), 142–164 (2011)
26. Minder, L., Sinclair, A.: The extended k-tree algorithm. Journal of Cryptology 25(2), 349–382 (2012)
27. Nandi, M.: Revisiting security claims of xls and copa. Cryptology ePrint Archive, Report 2015/444 (2015), https://eprint.iacr.org/2015/444
28. Naya-Plasencia, M.: How to Improve Rebound Attacks. In: Advances in Cryptology - CRYPTO 2011. LNCS, vol. 6841, pp. 188–205. Springer (2011)
29. Nielsen, M.A., Chuang, I.: Quantum computation and quantum information (2002)
30. Nikolic, I., Sasaki, Y.: Refinements of the k-tree Algorithm for the Generalized Birthday Problem. In: Advances in Cryptology - ASIACRYPT 2015. LNCS, vol. 9453, pp. 683–703. Springer (2015)
31. Schroeppel, R., Shamir, A.: A T = $O(2^{n/2})$, S = $O(2^{n/4})$ algorithm for certain np-complete problems. SIAM J. Comput. 10(3), 456–464 (1981)
32. Wagner, D.A.: A Generalized Birthday Problem. In: Advances in Cryptology - CRYPTO 2002. LNCS, vol. 2442, pp. 288–303. Springer (2002)

Appendix

A On Merging Trees

In this section, we elaborate on our use of Grover search, the probability of success in merging trees and possible extensions of the framework.

A.1 Grover Search as a Building Block

Sequence of Instances of Grover's Algorithm. Grover's algorithm is a probabilistic procedure. In particular, there is a (small) probability that the measurement yields an incorrect result. However, when we measure a result, we can immediately test if it is a good one, and discard it otherwise. Furthermore, our search spaces are all of exponential size in n, and the number of iterations in our Grover instances will always be an exponential in n (of the form $2^{\alpha_k n}$ for some constant α_k depending on k). The probability of error is always exponential in n and so, smaller than a constant in k. When running an exponential number of instances, if they run each after another, their results are tested. The probabilities of error do not multiply, so the error factor remains a constant in k.

Nested Instances of Grover's Algorithm. Some of our work also relies on nested instances of Grover's algorithm, as in [7]. In that case, we run a Grover search over some search space X, and to test an element $x \in X$, we do some computations requiring to run another Grover search. To analyze the success probability, we can do as if the inside Grover search result was measured (although it is always possible to defer this measurement at the end of the computation). When

it is measured, the success probability is exponential in n; but the number of iterations of the outer Grover search is also exponential in n. However, it is sufficient to repeat the inner search a logarithmic number of times (in n) to increase the success probability, so that it is constant for the whole procedure. Since we do not nest more than a constant (in k) number of Grover searches, we suffer at most from a polylogarithmic increase in complexity.

A.2 Ensuring a Success Probability of 1 with a Depth-First Tree Traversal

A node in the tree is a list L of 2^t ℓ-tuples which partially XOR to zero on v bits, and of their XOR. During the computation, a candidate x having also a zero-prefix of v bits will be matched against this node, expecting a XOR with more zeroes, say u new zeroes. If we are not at the root of the tree, we want at least one match (on average), so $u \leq t$. For each candidate to find a match, it suffices to ensure that the elements of L take all possible bit-strings in the u bits of the match.

This is a coupon collector problem with 2^u coupons, ensuring that up to logarithmic factor in n and constant in k the complexities remain unchanged.

While constructing L, each new element in this list draws a random coupon among 2^u; we want all coupons with high probability. Let T be the time to obtain all coupons, and in our case, the number of elements produced before our condition is met. For all β we have:

$$\mathbb{P}(T > \beta u 2^u) \leq (2^u)^{-\beta+1}$$

in particular, we crudely bound u by n and notice that:

$$\mathbb{P}(T > \beta n|L|) \leq (2^u)^{-\beta+1}$$

so by taking $\beta = 2$, we can see that, by putting an additional factor $2n$ on the size of all lists, the probability that the algorithm fails is lower than:

$$\sum_{\text{lists } L} \mathbb{P}(T > \beta n|L|)$$

which is negligible in n, as the value u is always a multiple of n, and the total number of lists in the tree is not more than k^2. Even in the limit case where k approaches $2^{\sqrt{n}}$, taking a slightly increased value of β is sufficient.

These considerations hold independently of the way list elements are produced. If producing an element is a probabilistic procedure which has a success probability of $p < 1$ (*e.g.* a run of Grover's algorithm), since elements are easy to check, the total cost is only multiplied by $\frac{1}{p}$. This is why, up to a logarithmic factor in n and a constant in k, the complexities are unchanged. In the rest of this paper, we will compute and optimize exact complexity exponents given as multiples of n, depending on k.

Sorting and random-access. When a list of size N is built, we consider that storing it in memory costs at least time N (to write it), regardless of the memory used (quantum or classical) and of the memory access authorized (random-access or not). The lists stored are supposed to be sorted (it takes time $\widetilde{\mathcal{O}}(N)$), and quantum and classical random access is performed in time $\mathcal{O}(\log N)$, so all these factors are outside our focus in this paper.

A.3 Other Extensions of Merging Trees.

Inspired by the classical works that extended the original k-tree algorithm of Wagner [15,3,16,30,26], we considered many other possible extensions to merging trees than the ones of Section 6. However, none of them seemed so far to give an actual quantum improvement, whether in time or memory.

Clamping. For example, *clamping* [6] corresponds exactly to taking a prefix of zeroes for the first leaf of a given node in the tree. This reduces memory usage, to the cost of an increase in time.

Chain-ends. One can replace some lists of single elements by lists of chain-ends, using Hellman tables [30]. Classically, the simplest is to replace all leaves in Wagner's original tree by such tables. It seems possible to translate this in our framework. Assume we have a node T with children $T_0, \ldots, T_{\ell-1}$, ordered by their number of nodes. Consider the last child T_i which is a leaf. Constraint all the other leaves to contain a single, random element (they are simply dismissed). We replace T_i by a list of chain-ends of some length. When performing the search for new elements of T, we test collision on T_i by recomputing a chain-end of the same length. If T_i is of length 2^v and contains chain-ends of chains of length 2^u, if T has a prefix of length t, then the classical time complexity is (we don't write the factors stemming from the other children and focus on the chain-ends):

$$2^{u+v} + \ldots + 2^{t-2u-v-\ldots}\,(2^u + \ldots)$$

since roughly, the table allows to find a collision on $2u + v$ bits, which is then matched against the other children (whose size has to be optimized accordingly).

Quantumly, iterating a function 2^u times still requires 2^u calls. In that case, the time complexity becomes roughly:

$$2^{u+v} + \ldots + 2^{(t-2u-v-\ldots)/2}\,(2^u + \ldots)$$

and we can remark that using *clamping* on u bits (so taking, for the first leaf, elements with a prefix of u zeroes) uses the same amount of memory but can only improve on the time complexity:

$$2^{u/2+v} + \ldots + 2^{(t-u-v-\ldots)/2}\left(2^{u/2} + \ldots\right) \quad .$$

The reason for that is that quantumly, finding elements with arbitrary prefixes is faster than iterating the function (which is not the case classically).

We have not studied the techniques of [16, Section 4], which use a similar idea in a different context.

B Quantum Algorithms for Problem 2

In this section, we give more details on quantum merging algorithms for Problem 2.

B.1 Quantum Low-qubits Algorithm for 5-xor

For $k = 5$, our algorithm has time complexity $2^{14n/45}$ and classical memory complexity $2^{7n/45}$. The optimized merging tree is depicted on Figure 11. The computation runs as follows:

- Build node T_0^2: a list of size $2^{7n/45}$ of elements with zero-prefix of $\frac{14}{45}n$ bits, using a sequence of $2^{7n/45}$ Grover searches in time:
$$2^{\frac{7}{45}n} \times 2^{\frac{1}{2}\times\frac{14}{45}n} = 2^{\frac{14}{45}n} \ .$$
- Build node T_2^1: a list of size $2^{2n/15}$ of collisions on $\frac{23}{45}n$ bits, using the list of T_0^2 as intermediate, in time:
$$2^{\frac{2n}{15}} \times \underbrace{2^{\frac{1}{2}\times\frac{2}{45}n}}_{\substack{\text{There remains}\\ \frac{2n}{45} \text{ bits to put to zero}}} \left(\underbrace{2^{\frac{1}{2}\times\frac{14}{45}n}}_{\substack{\text{Finding the}\\ \text{zero-prefix}}} + \underbrace{2^{\frac{7}{45}n}}_{\text{Lookup of } T_0^2} \right) = 2^{\frac{14}{45}n} \ .$$
Indeed, given an element with $\frac{14}{45}n$-bit zero prefix, we find a collision with T_0^2 on $\frac{7n}{45}$ more bits, and to obtain a collision on $\frac{23}{45}n$ bits there remains only $\frac{n}{45}$ iterations to perform.
- Build node T_0^1: a list of size $2^{2n/15}$ of elements with zero-prefix of $4n/15$ bits
- Build node T_1^1: a list of $2^{n/9}$ elements with zero-prefix of $2n/5$ bits
- Build node T_0^0: a single 5-xor on n bits. For each element with the good zero prefix, T_0^1 gives a partial collision on $2n/15$ more bits. At this point we have a collision on $2n/5$ bits. So there is an element in T_1^1 yielding a partial 3-xor on $5n/45$ more bits, hence $\frac{23n}{45}$ bits. And there is an element in T_2^1 yielding a partial 4-xor on $2n/15$ more bits, hence $\frac{29n}{45}$. There remain $\frac{16n}{45}$ bits to put to zero for a given element in the search space, so the Amplitude Amplification procedure needs $2^{\frac{8n}{45}}$ iterations.

  In each iteration, the setup requires $2^{2n/15}$, to reduce the search space to the elements of good zero-prefix. The test requires $2^{2n/15} + 2^{n/9} + 2^{2n/15}$ to go through the three children and find the partially colliding elements (we suppose that there is exactly one, see Section A.2). The total time is:
$$\underbrace{2^{\frac{8n}{45}}}_{\text{Iterations}} \left(\underbrace{2^{\frac{2n}{15}}}_{\substack{\text{Lookup of}\\ T_0^1}} + \underbrace{2^{\frac{n}{9}}}_{\substack{\text{Lookup}\\ \text{of } T_1^1}} + \underbrace{2^{\frac{2n}{15}}}_{\substack{\text{Lookup of}\\ T_2^1}} \right) = 2^{\frac{14}{45}n} \ .$$

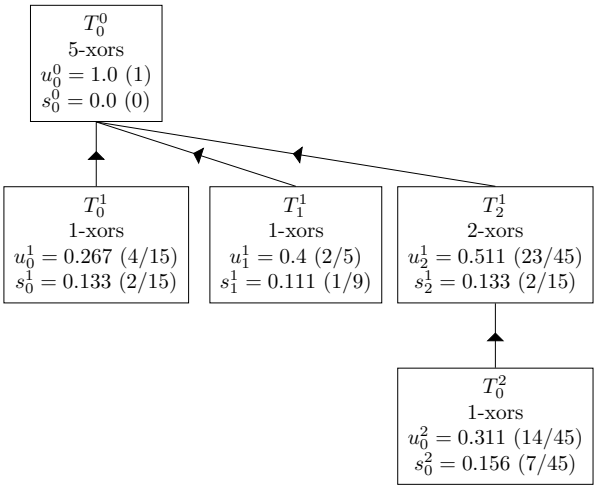

Fig. 11: Optimized Merging Tree for low-qubits 5-xor

B.2 Merging Tree Complexities for Problem 2

In Table 1, we give our results for solving Problem 2 (k-xor with many solutions and quantum oracle access). The table displays α_k instead of a time complexity $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$. In the left two columns, the classical complexities are displayed for comparison. In the next two columns, we give our results. On the right side of the table, we give them as fractions and compare with [17]. We highlight the improvements.

B.3 Proof of Optimality in the qRAM Case

We give the full proof of Theorem 1 in Section 5.2:

Theorem 1. *Let $k \geq 2$ be an integer and $\kappa = \lfloor \log_2(k) \rfloor$. The best quantum merging tree finds a k-xor on n bits in quantum time (and memory) $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$ where $\alpha_k = \frac{2^\kappa}{(1+\kappa)2^\kappa + k}$. The same method finds 2^{nc} k-xor with a quantum (time and memory) complexity exponent of $n \max\left(\alpha_k + 2\alpha_k c, c\right)$.*

Furthermore, for every k, the optimum is realized by $\mathcal{T}_k$.

Proof. We prove this by induction on k. Recall that we write the complexities in $\log_2$ and as multiples of n, so instead of writing $2^{ns_1} = 2^{ns_2}$ we write $s_1 = s_2$.

We consider a merging tree T that builds a single k-xor. The root has ℓ children denoted $T_0, \dots T_{\ell-1}$. Let $s = 0, s_0, \dots s_{\ell-1}$ be the size variables of $T, T_0, \dots T_{\ell-1}$ respectively and $u, u_0 = 0, u_1, \dots u_{\ell-1}$ be the size of zero-prefixes of $T, T_0, \dots T_{\ell-1}$ respectively. We also note $k_0, k_1, \dots k_{\ell-1}$ the number of nodes of $T_0, \dots T_{\ell-1}$ respectively, and we have $k_0 + \dots + k_{\ell-1} + 1 = k$. We order the children so that $k_0 \leq k_1 \leq \dots \leq k_{\ell-1}$ and $u_0 \leq \dots \leq u_{\ell-1}$.

Table 1: Best time complexity exponents of k-xor optimized merging trees for k up to 12, and comparison with the results from [17].

k	Classical Rounded	Classical Fraction	qRAM Rounded	Low-qubits Rounded	Comparison qRAM This paper	Comparison qRAM [17]	Comparing Low-qubits This paper	Comparing Low-qubits [17]
2	0.5	1/2	0.333	0.4	1/3	1/3	2/5	2/5
3	0.5	1/2	0.286	0.357	**2/7**	3/10	5/14	5/14
4	0.333	1/3	0.25	0.333	4/16	4/16	1/3	1/3
5	0.333	1/3	0.235	0.311	**4/17**	4/16	**14/45**	7/22
6	0.333	1/3	0.222	0.286	**4/18**	4/16	**2/7**	4/13
7	0.333	1/3	0.211	0.286	**4/19**	4/16	**2/7**	3/10
8	0.25	1/4	0.2	0.25	8/40	8/40	1/4	1/4
9	0.25	1/4	0.195	0.25	**8/41**	1/5	1/4	1/4
10	0.25	1/4	0.190	0.25	**8/42**	1/5	1/4	1/4
11	0.25	1/4	0.186	0.25	**8/43**	1/5	1/4	1/4
12	0.25	1/4	0.182	0.222	**8/44**	1/5	**2/9**	1/4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Remark 1. For simplification we suppose that we build a single k-xor. In general, when we want to produce 2^{nc} k-xors for c small enough, the last Grover step will be repeated 2^{nc} times. If we rewrite carefully the cost constraints, we remark that all is optimized as if we computed a single k-xor on $n + 2nc$ bits instead of n bits, hence the updated complexity exponent. When c is too big, though, we fall back on classical search.

We first remark that $u_0 = 0$: if we force a non-empty zero-prefix for the first child, we have to pay a setup of cost $2^{u_0/2}$ at each iteration of Grover search at the root T; whereas we could remove the common u_0 bits of prefix from all subtrees, produce them in less time, and do $2^{u_0/2}$ more iterations. Since $u_0 = 0$, the first child also has $k_0 = 1$: producing k_0-tuples gives no advantage over single elements. Its cost is s_0, equal to its size.

By the constraints of Section 4.5, we have:

$$u_0 = 0, u_1 = s_0, u_2 = s_1 + s_0, \ldots, u_{\ell-1} = \sum_{i=0}^{\ell-2} s_i \ .$$

We can use the recurrence hypothesis on all subtrees, since $k_i < k$. The cost of building each of them is: $u_i\alpha_{k_i} + 2s_i\alpha_{k_i}$ for $i > 1$. When the tree is optimized, we expect all these costs to become equal. We have:

$$\forall i \geq 1, u_i\alpha_{k_i} + 2(u_{i+1} - u_i)\alpha_{k_i} = s_0 \implies u_{i+1} = \frac{u_i}{2} + \frac{s_0}{2\alpha_{k_i}} \ . \tag{1}$$

If we produce 2^{nc} k-xors, so if the root node is a list of size 2^{nc}, it can be shown additionally that as long as $\alpha_k + 2\alpha_k c < c$, then $\alpha_{k_i} + 2\alpha_{k_i} s_i < s_i$. In

other words, the subtrees remain in the regime in which 2^{nc} k_i-xors on n bits cost $2^{2\alpha_{k_i} nc}$ times the price of a single one, and not the regime in which 2^{nc} k_i-xors on n bits cost 2^{nc} (in which case all the lists are built classically, except for the leaf nodes).

By an induction on i in (1), we can expand $u_{\ell-1}$ depending on s_0 and the α_{k_i}: $u_1 = s_0$, $u_2 = \frac{u_1}{2} + \frac{s_0}{2\alpha_{k_1}} = s_0 \left(\frac{1}{2} + \frac{1}{2\alpha_{k_1}}\right)$, and:

$$u_{\ell-1} = s_0 \left(\frac{1}{2^{\ell-2}} + \sum_{i=1}^{\ell-2} \frac{1}{2^{\ell-1-i}\alpha_{k_i}}\right) \quad . \tag{2}$$

All of this still comes from the tree constraints and the equality of the cost of all nodes. For the last child, we also have $u_{\ell-1}\alpha_{k_{\ell-1}} + 2s_{\ell-1}\alpha_{k_{\ell-1}} = s_0$ hence

$$s_{\ell-1} = \frac{1}{2\alpha_{k_{\ell-1}}} s_0 - \frac{u_{\ell-1}}{2} \quad . \tag{3}$$

Furthermore, the cost of subtree T_0, which is s_0, is equal to the cost of the final Grover search, which is $\frac{1}{2}(1 - s_{\ell-1} - u_{\ell-1})$ as shown in Section 4.5. By replacing $u_{\ell-1}$ and $s_{\ell-1}$ by their respective expressions (3) and (2), we obtain:

$$s_0 = \frac{1}{2}(1 - s_{\ell-1} - u_{\ell-1}) \implies 1 = 2s_0 + s_{\ell-1} + u_{\ell-1}$$

$$\implies 1 = s_0 \left(2 + \frac{1}{2^{\ell-1}} + \sum_{i=1}^{\ell-1} \frac{1}{2^{\ell-i}\alpha_{k_i}}\right) \tag{4}$$

We note $A(\alpha_{k_1}, \ldots, \alpha_{k_{\ell-1}}) = \left(2 + \frac{1}{2^{\ell-1}} + \sum_{i=1}^{\ell-1} \frac{1}{2^{\ell-i}\alpha_{k_i}}\right)$, hence the final complexity exponent will be $s_0 = 1/A$.

A Detour by the Multiple-Candidates Case. Before determining the constraints of Section 4.5, we defined our merging trees in more generality, by allowing that a child list yields multiple candidates. Assume for simplicity that we are in the qRAM case. In the exhaustive search operation that is performed by the node T, we test a new element x. During this test, we find successive *candidates*: the first child node yields a $x \oplus x_0$ with some bits to zero, the second child node yields $x \oplus x_0 \oplus x_1$ with even more bits to zero, etc. We assumed that there always was *exactly* one candidate at each step, but each list could actually yield 2^{c_i} of them, with new variables $c_i, 0 \leq i \leq \ell - 1$.

For example, if there are strictly less zeroes in L_1 than elements in L_0, then L_0 yields $c_i > 0$ candidates, that all collide with L_0 on the bit-positions zeroed in L_1. On the contrary, if there are more, hence $c_i < 0$, then we find a candidate (and move on to the next lists) only once over 2^{-c_i}. Inside the final Grover search, each current candidate requires a query to the next list, so although all

queries cost $\mathcal{O}(1)$, the test costs:

$$\max\left(\underbrace{\max(c_0,0)}_{\substack{\text{Number of}\\ \text{queries to } T_1}}\ ,\ \underbrace{\max(c_0,0)+\max(c_1,0)}_{\text{Number of queries to } T_2}\ ,\ \ldots,\ \underbrace{\sum_{i=0}^{\ell-2}\max(c_i,0)}_{\substack{\text{Number of queries to}\\ T_{\ell-1}}}\right)$$

the reason for the "inner" max in $\max(c_0,0)$ is that, even if a list yields less than one candidate on average and the next one more than one; we must handle all cases in superposition, so all happens as if each list yielded at least one candidate (so we replace c_i by $\max(c_i,0)$).

The relation between the u_i and s_i is also changed: we have $s_i = c_i + (u_{i+1} - u_i)$, by definition of c_i, regardless of its sign, so $u_{i+1} = \frac{u_i}{2} + \frac{s_0}{2\alpha_{k_i}} - c_i$ and finally:

$$u_{\ell-1} = \sum_{i=0}^{\ell-2} s_i = s_0\left(\frac{1}{2^{\ell-2}} + \sum_{i=1}^{\ell-2}\frac{1}{2^{\ell-1-i}\alpha_{k_i}}\right) - \sum_{i=0}^{\ell-2} c_i$$

and:

$$1 - s_{\ell-1} - u_{\ell-1} + 2\sum_{i=0}^{\ell-2}\max(c_i,0) = 2s_0$$

so we can adapt (4) by taking into account the c_i, and keeping the same expression $A(\alpha_{k_1},\ldots,\alpha_{k_{\ell-1}})$:

$$1 + 2\sum_{i=0}^{\ell-2}\max(c_i,0) = s_0\left(2 + \frac{1}{2\alpha_{k_{\ell-1}}}\right) + \frac{u_{\ell-1}}{2} = s_0 A - \frac{1}{2}\sum_{i=0}^{\ell-2} c_i \qquad (5)$$

since our intention is to minimize s_0 (the final complexity exponent), we remark that all the c_i should be equal to zero. Having more candidates increases the cost of the inner tests, while giving only a poor improvement on the number of iterations of the search. Having less increases the number of iterations, for a constant test cost. None of these situations improve.

Back to the Proof. What remains to do is to choose the k_i so that the quantity

$$A(\alpha_{k_1},\ldots,\alpha_{k_{\ell-1}}) = \left(2 + \frac{1}{2^{\ell-1}} + \sum_{i=1}^{\ell-1}\frac{1}{2^{\ell-i}\alpha_{k_i}}\right)$$

becomes maximal, and the complexity exponent $s_0 = 1/A$ minimal. We now single out the term depending on $k_{\ell-1}$ in this complexity and rewrite:

$$A(\alpha_{k_1},\ldots,\alpha_{k_{\ell-1}}) = 1 + \frac{1}{2}\left(2 + \frac{1}{2^{\ell-2}} + \sum_{i=1}^{\ell-2}\frac{1}{2^{\ell-i-1}\alpha_{k_i}}\right) + \frac{1}{2\alpha_{k_{\ell-1}}}$$

in which we recognize $A(\alpha_{k_1}, \ldots, \alpha_{k_{\ell-2}})$. Our recurrence hypothesis supposes that the best merging tree for $k' < k$ gives a complexity exponent $\alpha_{k'}$ with a certain formula. So the term $A(\alpha_{k_1}, \ldots, \alpha_{k_{\ell-2}})$, which corresponds to the complexity of a $k-k_{\ell-1}$ merging tree, must be smaller than $\frac{1}{\alpha_{k_{\ell-2}}}$. In other words, we rewrote the complexity of our k-merging tree so that it depends on the complexity of a $k - k_{\ell-1}$ merging tree, which is bounded by our recurrence hypothesis.

We have:

$$A(\alpha_{k_1}, \ldots, \alpha_{k_{\ell-1}}) \leq 1 + \frac{1}{2\alpha_{k-k_\ell}} + \frac{1}{2\alpha_{k_\ell}} \quad .$$

and we obtain an equality by choosing the tree structure according to the optimal $k - k_\ell$ merging. Our recurrence hypothesis also states that α_k is a strictly decreasing function of k. Hence the sum $\frac{1}{2\alpha_{k-k_\ell}} + \frac{1}{2\alpha_{k_\ell}}$ becomes optimal when $k - k_\ell$ is close to k_ℓ. In particular cases (when it is impossible to cut exactly in halves), there may be two choices of equivalent complexity. However, we can still write that this sum is smaller, for every k_ℓ, to the sum with $k_\ell = k - \lfloor k/2 \rfloor$. This precise choice gives the tree structure $\mathcal{T}_k$ described above. To finish the recurrence, we remark that the $\lfloor \log_2 \rfloor$ is the same for both $\lfloor k/2 \rfloor$ and $k - \lfloor k/2 \rfloor$, and equal to $\kappa - 1$ in both cases (this is a simple case disjunction whether k is a multiple of 2 or not). So we have:

$$\frac{1}{\alpha_k} = 1 + \frac{(1+\kappa-1)2^{\kappa-1} + k - \lfloor k/2 \rfloor}{2^\kappa} + \frac{(1+\kappa-1)2^{\kappa-1} + \lfloor k/2 \rfloor}{2^\kappa} = \frac{2^\kappa(1+\kappa) + k}{2^\kappa}$$

which is the expected result.

□

B.4 Optimization on an Example: 11-xor

The optimization of 11-xor is given below. Table 2 contains the constraints.

C Quantum Algorithms for Problems 1, 4 and 3

In this section, we give more details of the algorithms for the three other problems considered.

C.1 Unique k-xor Algorithms

We give the detail of quantum 3-xor algorithms with a domain size of $2^{n/3}$ and a single solution. The classical time complexity of this problem is $2^{2n/3}$.

Algorithms 5 and 6 differ only in that the latter allows access to a quantum oracle (which is the case for multiple-encryption), while the former supposes that the data is given classically. Both have a time complexity of $\mathcal{O}\left(2^{n/3}\right)$ which is necessarily optimal. Algorithm 6 uses only $2^{n/6}$ memory, while Algorithm 5 requires $2^{n/3}$ memory.

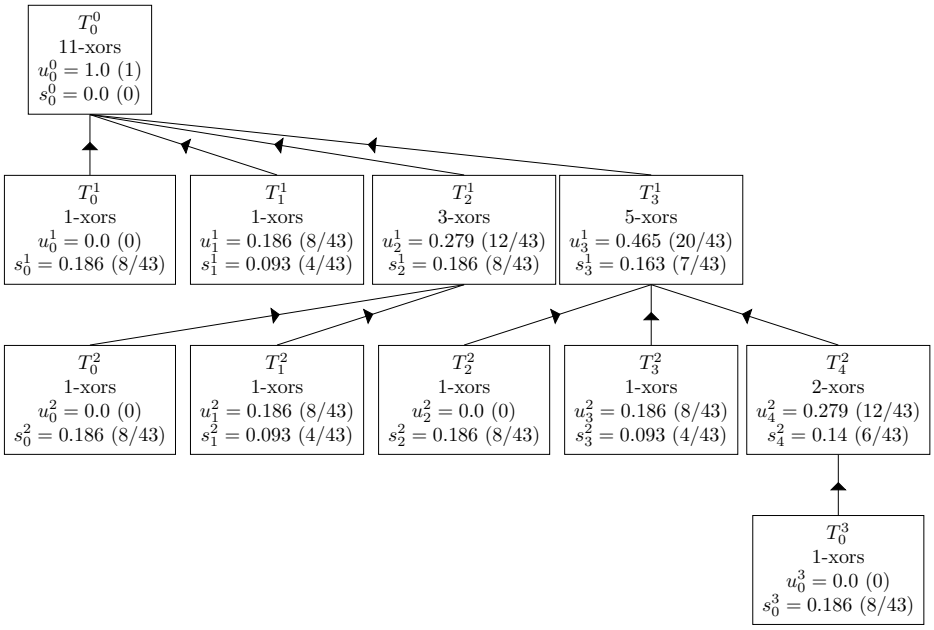

Fig. 12: 11-xor optimization of the tree $\mathcal{T}_{11}$

Table 2: Sequence of constraints on which we optimize the tree $\mathcal{T}_{11}$

Node name	Structural constraints	Cost
T_0^3		$c_0^3 = s_0^3 + \frac{u_0^3}{2}$
T_4^2		$c_4^2 = s_4^2 + \frac{1}{2}\left(u_4^2 + u_0^3 - u_0^3 - s_0^3\right)$
T_3^2		$c_3^2 = s_3^2 + \frac{u_3^2}{2}$
T_2^2		$c_2^2 = s_2^2 + \frac{u_2^2}{2}$
T_3^1	$u_4^2 = s_3^2 + u_3^2$ $u_3^2 = s_2^2 + u_2^2$	$c_3^1 = s_3^1 + \frac{1}{2}\left(u_3^1 + u_4^2 - u_2^2 - s_2^2\right)$
T_1^2		$c_1^2 = s_1^2 + \frac{u_1^2}{2}$
T_0^2		$c_0^2 = s_0^2 + \frac{u_0^2}{2}$
T_2^1	$u_1^2 = s_0^2 + u_0^2$	$c_2^1 = s_2^1 + \frac{1}{2}\left(u_2^1 + u_1^2 - u_0^2 - s_0^2\right)$
T_1^1		$c_1^1 = s_1^1 + \frac{u_1^1}{2}$
T_0^1		$c_0^1 = s_0^1 + \frac{u_0^1}{2}$
T_0^0	$s_0^0 = 0$, $u_0^0 = 1$ $u_3^1 = s_2^1 + u_2^1$ $u_2^1 = s_1^1 + u_1^1$ $u_1^1 = s_0^1 + u_0^1$	$c_0^0 = s_0^0 + \frac{1}{2}\left(u_0^0 + u_3^1 - u_0^1 - s_0^1\right)$

The time complexity of Algorithm 5 is:

$$2^{n/3} + \underbrace{2^{n/6}}_{\substack{\text{Search on} \\ x}} \times \underbrace{2^{n/6}}_{\substack{\text{Search on} \\ z}} .$$

Algorithm 5 Quantum algorithm for the unique 3-xor problem, with qRAM, without oracle access.

Store all the elements in a list L_0 (qRAM-accessible)
for $x \in L_0$ **do**
  for $y \in L_0$ **do**
    Find $z \in L_0$ with $x \oplus y \oplus z = 0|*$
    If $x \oplus y \oplus z = 0$, return this result.
  end for
end for

Algorithm 6 Quantum algorithm for the unique 3-xor problem (or 3-encryption), with qRAM and oracle access

for $u \in \{0,1\}^{n/6}$ **do**
  Build a list L_0 of $2^{n/6}$ elements with prefix u, using Grover search in time $2^{n/4}$
  for $2^{n/6}$ repetitions **do**
    Build a list L_1 of $2^{n/6}$ elements by querying H
    for $x' \in \{0,1\}^{n/3}$ **do**
      Query $x = H(x')$
      Find $y \in L_0$ with $z \oplus y = u|*$
      If there exists $z \in L_1$ such that $x \oplus y \oplus z = 0$, return.
    end for
  end for
end for

The time complexity of Algorithm 6 is:

$$\underbrace{2^{n/12}}_{\text{Search on } u}\left(2^{n/4}+2^{n/12}\left(\underbrace{2^{n/6}}_{\substack{\text{Produce}\\ L_1}}+\underbrace{2^{n/6}}_{\substack{\text{Search on}\\ x'}}\right)\right).$$

We also give the detail of our quantum algorithm for unique 5-xor (or 5-encryption). The time complexity of Algorithm 7 is $2^{n/10}\left(2^{n/5}+2^{n/5}\right)=2^{3n/10}$. It uses a memory $2^{0.2n}$.

In general, to reach the time $\widetilde{\mathcal{O}}\left(2^{0.3n}\right)$, our algorithms for unique k-xor require an amount of $2^{0.2n}$ qRAM, so they do not improve over the 4-xor quantum walk with respect to the time-memory product. The time-memory product makes sense here, as the entire memory is quantum RAM, and may require active hardware (see [20] or [23]). It is merely a change in the optimization goal, the time-memory product being (in $\log_2$) the sum of two variables. We obtain the results of Table 3. They are difficult to describe, especially since the best algorithms with this respect are not the best algorithms in time. For example

Algorithm 7 Quantum algorithm for the unique 5-xor problem (or 5-encryption), with or without oracle access. We consider a single list; the case of 5 separate lists is similar.

Store all $2^{n/5}$ elements (input list L) in qRAM
for $u \in \{0,1\}^{n/5}$ **do**
    Build a list L_0 of $2^{n/5}$ collisions with prefix u, in time $2^{n/5}$, by using L as intermediate
    for $x \in L$ **do**
        for $y \in L$ **do**
            Find $z \in L$ such that $x \oplus y \oplus z$ has prefix u
            If there exists $t \oplus t' \in L_0$ such that the whole 5-xor is zero, return.
        end for
    end for
end for

Algorithm 8 Translation of algorithm 7 to the 5-encryption setting.

Input: 5 plaintext-ciphertext pairs (P_i, C_i) which form each an n-bit condition
Output: a sequence of 5 n-bit keys k_0, k_1, k_2, k_3, k_4 such that $E_{k_4} \circ E_{k_3} \circ E_{k_2} \circ E_{k_1} \circ E_{k_0}(P_i) = C_i$
for $X \in \{0,1\}^n$ **do** ▷ Repetition of the last child
    Compute $E_{k_4}^{-1}(C_1)$ for all k_4
    Compute $E_{k_3}(X)$ for all k_3
    Build the list of all pairs k_3, k_4 such that $E_{k_4} \circ E_{k_3}(X) = C_1$, in time 2^n
    Compute $E_{k_3}^{-1} \circ E_{k_4}^{-1}(C_2)$ for all these pairs
    Compute $E_{k_2}^{-1}(X)$ for all k_2
    for any k_1 **do** ▷ Repetition of the third-to-last child
        for any k_0 **do** ▷ Root exhaustive search
            Find k_2 such that $E_{k_1} \circ E_{k_0}(P) = E_{k_2}^{-1}(X)$
            Find k_3, k_4 in the list of precomputed pairs such that $E_{k_3}^{-1} \circ E_{k_4}^{-1}(C_2)$ is equal to $E_{k_2} \circ E_{k_1} \circ E_{k_0}(P_2)$
            The tuple k_0, k_1, k_2, k_3, k_4 now enciphers P_1 to C_1 and P_2 to C_2
            If it also matches on the other plaintext-ciphertext pairs, return.
        end for
    end for
end for

for $k = 10$, we have an algorithm running in time $\widetilde{\mathcal{O}}\left(2^{0.35n}\right)$ using $2^{0.1n}$ memory. Our point of interest lies at $k = 12$, with time $\widetilde{\mathcal{O}}\left(2^{n/3}\right)$ and memory $2^{n/12}$, where the product is minimal among the instances studied. But we do not know whether it is a minimum over all k.

Table 3: Our quantum time and memory complexities for Problems 4 and 3, given as $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$. We emphasize in the first column the improvements with respect to the previous quantum algorithms known.

k	Our best time		Corresponding memory		Best t.-m. product	
	As fraction	Rounded	As fraction	Rounded	As fraction	Rounded
3	**1/3**	**0.3333**	1/3	0.3333	2/3	0.6667
4	5/16	0.3125	1/4	0.25	9/16	0.5625
5	**3/10**	**0.3**	1/5	0.2	1/2	0.5
6	1/3	0.3333	1/6	0.1667	1/2	0.5
7	**9/28**	**0.3214**	1/7	0.1429	13/28	0.4643
8	5/16	0.3125	1/8	0.125	7/16	0.4375
9	**11/36**	**0.3056**	1/6	0.1667	4/9	0.4444
10	**3/10**	**0.3**	1/5	0.2	9/20	0.45
11	**7/22**	**0.3182**	3/22	0.1364	5/11	0.4545
12	5/16	0.3125	1/6	0.1667	5/12	0.4167
13	**4/13**	**0.3077**	2/13	0.1538	23/52	0.4423
14	**17/56**	**0.3036**	5/28	0.1786	13/28	0.4643
15	**3/10**	**0.3**	1/5	0.2	7/15	0.4667

C.2 Merging Tree Complexities for Problem 1

In Table 4, we give the exponents obtained for Problem 1 (k-xor with classical inputs).

Table 4: Complexity of the k-xor problem with classical inputs, compared with quantum oracle access, as $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$.

k	Classical α_k		Without oracle		With oracle	
	As fraction	Rounded	Rounded	As fraction	Rounded	As fraction
3	1/2	0.5	0.3333	1/3	0.2857	2/7
4	1/3	0.3333	0.2857	2/7	0.25	1/4
5	1/3	0.3333	0.25	1/4	0.2353	4/17
6	1/3	0.3333	0.25	1/4	0.2222	2/9
7	1/3	0.3333	0.2353	4/17	0.2105	4/19
8	1/4	0.25	0.2222	2/9	0.2	1/5
9	1/4	0.25	0.2	1/5	0.1951	8/41
10	1/4	0.25	0.2	1/5	0.1905	4/21
11	1/4	0.25	0.2	1/5	0.186	8/43
12	1/4	0.25	0.1951	8/41	0.1818	2/11
13	1/4	0.25	0.1905	4/21	0.1778	8/45
14	1/4	0.25	0.1818	2/11	0.1739	4/23