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

María Naya-Plasencia, André Schrottenloher

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 or Generalized Birthday Problem aims at finding, given k lists of bit-strings, 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. If the lists are bounded (of the same size) and such that there is a single solution, the *dissection algorithms* of Dinur *et al.* (CRYPTO 2012) improve the memory usage over a simple meet-in-the-middle.

In this paper, we study quantum algorithms for the k-xor problem. With unbounded lists and quantum access, we improve previous work by Grassi *et al.* (ASIACRYPT 2018) for almost all k. Next, we extend our study to lists of any size and with classical access only.

We define a set of "merging trees" which represent the best known strategies for quantum and classical merging in k-xor algorithms, and prove that our method is optimal among these. 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 also when considering modular additions instead of bitwise xors.

This framework enables us to give new improved quantum k-xor algorithms for all k and list sizes. Applications include the subset-sum problem, LPN with limited memory and the multiple-encryption problem.

Keywords: Generalized Birthday Problem, quantum cryptanalysis, list-merging algorithms, k-list problems, approximate k-list problem, multiple encryption, MILP, LPN, subset-sum.

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 $H(x) = H(y)$. It is well-known that $\Omega(2^{n/2})$ classical queries are necessary and sufficient. In a seminal paper, Wagner [34] generalized a method credited to Camion and Patarin [15] to solve a variant of this problem for k-tuples:

> 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 $x_1 \oplus x_2 \oplus \ldots \oplus x_k = 0$.

Although Wagner studied the case of unbounded lists, many cryptographic applications are concerned with lists of limited size. For example, if the lists (of uniformly random n-bit strings) have size $2^{\lfloor n/k \rfloor}$, we expect a single solution with constant probability. The best classical algorithms for this case are given in [18], and apply *e.g.* to the multiple-encryption or subset-sums problems. Alternatively, if the lists have size $2^{\lfloor n/(k-1) \rfloor}$ we may want to find all the expected $2^{\lfloor n/(k-1) \rfloor}$ solutions.

Extension to Other Operations. We choose to focus on the bitwise XOR operation $\oplus$ for simplicity. In all algorithms studied throughout this paper, it can be replaced by modular additions. We provide more details in Appendix A.3.

Classical Complexity of k-xor. Intuitively, increasing k can only make the problem easier on average, since new degrees of freedom are available. The optimal 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 [34] 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 $2^{n/(\lfloor \log_2(k) \rfloor + 1)}$.

Quantum Complexity. The optimal quantum query complexity of k-xor is known to be $\Omega\left(2^{n/(k+1)}\right)$ [5]. In [20] some quantum algorithms for the solving the k-xor problem with quantum oracle access are given. For a general k, a time complexity of $\widetilde{\mathcal{O}}\left(2^{n/(\lfloor \log_2(k) \rfloor + 2)}\right)$ is obtained in the MNRS quantum walk framework [28]. As for Wagner's algorithm, the exponent decreases only at powers of 2. However, the authors also observed an exponential separation between the quantum collision and 3-xor time complexities. While collision search requires provably $\Omega(2^{n/3})$ quantum queries, they present a 3-xor algorithm running in time $\mathcal{O}\left(2^{3n/10}\right)$. A natural question is whether this extends to all k.

Furthermore, this previous work for general k covers only the case of unbounded lists. As highlighted above, in many applications we would like to consider a general k *and* lists of bounded size, as in [18,29].

This paper. In this work, we first answer the open questions stated in [20], which were far for intuitive or trivial as explained in section 3. We introduce for this the "merging trees", that describe in a systematic way merging strategies to solve the quantum k-xor problem. This enables us to reach better exponential time complexities than [20], with exponents that decrease strictly at each new value of k. 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.

While [20] studied the problem with quantum oracle access, we extend our framework to classically given lists *and* lists of limited size, up to the case where k lists of size only $2^{n/k}$ are given as input, improving the best algorithms for most values of k. We give the first quantum k-list algorithms applicable for all bicomposite problems as defined in [18]. We obtain also the first quantum time-memory product below $2^{n/2}$ for a generic k-list problem with lists of size $2^{n/k}$.

We provide several applications of these algorithms, improving the best known quantum algorithms for subset sums, the BKW algorithm, multiple-ecryption and the approximate k-list problem.

Outline. 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 algorithmic results. Sections 4 and 5 concern the case of unbounded lists. 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 [20]. Next, in Section 6, we extend to limited input domains, *i.e.* smaller lists. Finally, in Section 7, we give some applications, using our new k-list algorithms as black boxes: subset-sums, LPN, the approximate k-list and multiple-encryption problems. 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 Problem

All algorithms in this paper have exponential time complexities in n, written $\widetilde{\mathcal{O}}(2^{\alpha_k n})$ 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 uniformly 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 [34], in which the sizes of the lists is arbitrary, and not a concern. 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). The *oracle* version of this problem is as follows.

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 both will be used in the rest of this paper.

Problem 2 is the one studied in [20], 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.

Finally, 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".

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 uniformly random 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 use 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 [31].

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 $\alpha \left|0\right\rangle + \beta \left|1\right\rangle$, where $\left|0\right\rangle, \left|1\right\rangle$ 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 $\left|0\right\rangle$; then a sequence of unitary operators (formed of operators known as *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 $\alpha \left|0\right\rangle + \beta \left|1\right\rangle$ gives 0 with probability $|\alpha|^2$ while setting the state to $\left|0\right\rangle$.

A widely known example of quantum algorithm is Grover's algorithm [21]. From a uniform superposition over a search space X, it creates the superposition over the subset $G = \{x \in X, f(x) = 1\}$ for some function f, assuming that a superposition oracle for f is given: $O_f(\left|x\right\rangle \left|b\right\rangle) = \left|x\right\rangle \left|b \oplus f(x)\right\rangle$. As this procedure consists in iterating $\sqrt{|X|2^{-t}}$ times the same unitary, we speak of "iterations".

Lemma 1 (Grover Search, from [21]). *Let X be a search space, whose elements are represented on $\lceil \log_2(|X|) \rceil$ qubits, such that the uniform superposition $\frac{1}{\sqrt{|X|}} \sum_{x \in X} \left|x\right\rangle$ is computable in $\widetilde{\mathcal{O}}(1)$ time. Assume that we can implement a superposition oracle O_f for f in $\widetilde{\mathcal{O}}(1)$ time. Let $G = \{x \in X, f(x) = 1\}$. 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 $\widetilde{\mathcal{O}}\left(\sqrt{|X|}\right)$.*

Grover search is known to be optimal when the test f is a black-box oracle [6].

Amplitude Amplification. A generalization of Grover search given in [12] 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 $\widetilde{\mathcal{O}}\left(\sqrt{|X|2^{-t}}(|\mathcal{A}| + |O_f|)\right)$.

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.2.

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.* [24]), or detailed quantum gate counts.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.

qRAM Models. Classical random-access memory authorizes a constant-time access to memory cell whose indices are known only at runtime. However, during a quantum computation, the index register of such a query, since it depends on previous computations, is likely to be in superposition. This is why many quantum algorithms require quantum RAM.

A qRAM authorizes *superposition* access to its contents, using so-called "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. Using qRAM gates, it is possible to obtain quantum data structures with fast lookups (for example the combination of a skip list and a hash table in [2] or the radix trees of [7]). The access time is generally logarithmic and often neglected as a global multiplicative factor.

In this paper, we will extensively refer to three settings.

- "Low-qubits": the quantum computation uses only $\mathcal{O}(n)$ qubits and there are no qRAM gates. The quantum computer can still make use of a classical memory of exponential size, by performing classically controlled operations. This model was already considered in [20] and [16].
- QACM (quantum-accessible classical memory): there are qRAM gates, but the data accessed must be classical. This is the model required by the collision-finding algorithm of [13] or the QBKW algorithm of [19]. Some authors [27] consider it more relevant than the QAQM model.
- QAQM (quantum-accessible quantum memory): the quantum computation can use as many qubits as needed. The data accessed in superposition can be quantum. This model is obviously the most powerful. The unique collision-finding algorithm of [2] and the quantum algorithms for subset-sum of [7,23] require QAQM, as do all cryptographic applications of the MNRS quantum walk framework [28].

2.3 Overview of Previous Related Work

Classical Algorithms for the k-xor Problem. In Section 4, we will describe in detail Wagner's algorithm [34], 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). Many subsequent works have improved the memory consumption and given new trade-offs [8,32].

Minder and Sinclair [29] study the success probability and 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. They also perform a precise estimation of the success probability of Wagner's algorithm.

In [18], the authors study a family of *bicomposite* problems with a single solution, which include hard knapsacks, multiple-encryption, and k-xor with a single solution. They generalize the technique of Schroeppel and Shamir [33] 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.

Later on, more generalized frameworks have appeared, like [3] in the context of the Short Integer Solution problem, or [17], 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 [32], with dissection [33,18]. Although we have considered various potential improvements, our best algorithms for k-xor combine merging (as done by Wagner in [34]) and guessing intermediate values (as done in [18]), which is why we focus only on these techniques.

Quantum Algorithms for $k = 2$. The first algorithm to find quantum collisions was found by Brassard, Høyer and Tapp in 1998 [14,13]. With a two-to-one 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 [1] and extended to random functions [35]. This corresponds to the 2-xor problem with no bound on the list size. This algorithm also requires a QACM of size $2^{n/3}$.

When all 2^n outputs of H are distinct, except two of them, Ambainis' celebrated algorithm [2], based on a quantum walk, finds the pair in time $\widetilde{\mathcal{O}}\left(2^{2n/3}\right)$ using $2^{2n/3}$ QAQM. This corresponds to the 2-xor problem with a single solution. In the QACM model, there is, to date, no quantum algorithm with better time than the classical meet-in-the-middle.

Chailloux *et al.* [16] showed that the unbounded 2-xor problem could be solved in quantum time $\mathcal{O}\left(2^{2n/5}\right)$ in the low-qubits setting. The uses a classical memory of size $2^{n/5}$. Indeed, a superposition query to a QACM of size $2^{n/5}$ can be emulated by $2^{n/5}$ sequential quantum computations. The cost of these queries is mitigated by the fact that the algorithm makes only $2^{n/5}$ of them.

Quantum Algorithms for bigger k. Given a random function $H : \{0,1\}^n \to \{0,1\}^n$, the classical (information-theoretic) query lower bound of the k-xor problem is $\Omega(2^{n/k})$. The quantum query lower bound is $\Omega(2^{n/(k+1)})$ [5].

Unbounded Domain Size. Grassi *et al.* [20] proposed quantum algorithms for solving the k-xor problem with a quantum oracle for a random function $H : \{0,1\}^n \to \{0,1\}^n$, hence in the case of unbounded lists, as in [34]. They proposed a quantum analogue of Wagner's algorithm based on a quantum walk, running in the QAQM model, in time $\widetilde{\mathcal{O}}\left(2^{n\frac{1}{2+\lfloor \log_2 k \rfloor}}\right)$ and obtained some quantum time speedups in the low-qubits model. They also obtained a 3-xor QACM algorithm of quantum time complexity $\widetilde{\mathcal{O}}\left(2^{0.3n}\right)$, with an exponential improvement over quantum collision search. In this paper, we subsume and improve all these results. Notably, our new algorithms in this case require QACM only.

Restricted Domain and Unique k-xor. To the best of our knowledge, the k-xor problem with limited domain size, including Problem 3, has never been studied for a general k from a quantum algorithmic perspective. For $k = 4$, a quantum walk algorithm (originally designed for solving subset sums) is given in [7]. It solves Problem 3 in time $\widetilde{\mathcal{O}}\left(2^{0.3n}\right)$, using $\widetilde{\mathcal{O}}\left(2^{0.2n}\right)$ QAQM. This represents an exponential quantum time and memory improvement with respect to $k = 2$. However, for other values of k, *e.g.* $k = 5$, we must revert to a simple meet-in-the-middle strategy using Ambainis' algorithm.

Moreover, while Ambainis' algorithm gives a general meet-in-the-middle result, the 4-list algorithm of [7] is not a general 4-dissection algorithm; it does not apply to 4-encryption (we will explain this in Section 7).

3 Summary of our Main Results

In this section we summarize the optimal time complexities, *in our merging tree framework*, for solving Problems 1, 2, 3 and 4, with XORs and modular additions. 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 [20]. 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 $k \leq 20$. From these results, we were able to understand the optimal methods and extrapolate the results given below.

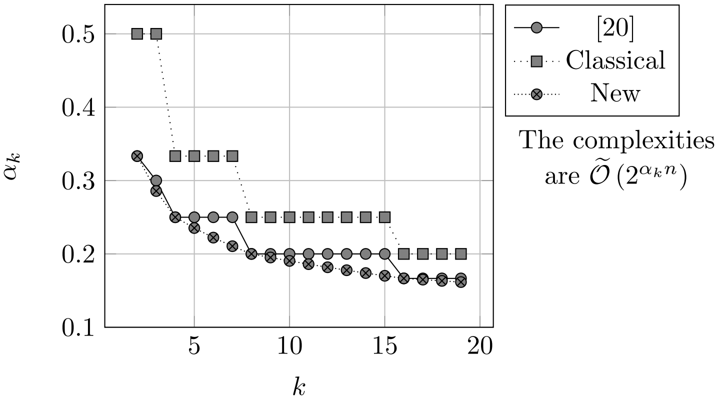
New quantum algorithms for LPN, subset-sums, multiple-encryption and the parity check problem. Whenever a classical algorithm makes use of a black-box k-xor procedure, we can replace this inner machinery with a quantum merging algorithm and optimize the strategy using MILP. We have identified various cryptographic applications of our framework. However, we defer the details to Section 7 and concentrate here only on the black-box k-xor problems.

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

3.1 Quantum Algorithms for Problem 2

In the QACM setting, we prove Theorem 1 and, answering one of the open questions of [20], show that the time complexity exponent of our method decreases strictly for each k (see Figure 1 for a comparison).

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



The complexities are $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$

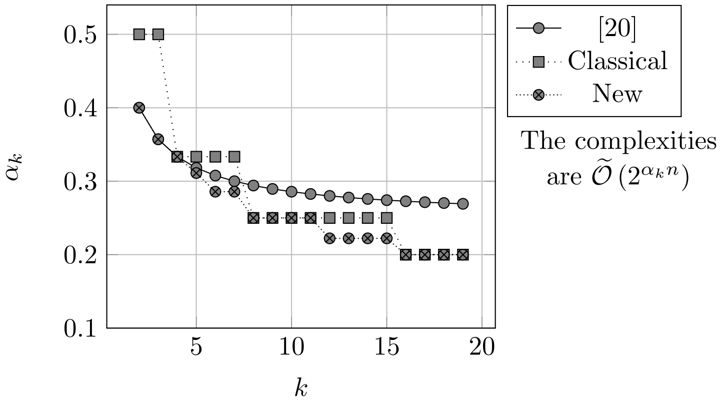
Fig. 1: Comparison of time complexity exponents between the classical case, the algorithms of [20] and our new results, in the QACM setting.

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 [20] (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:*

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

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



The complexities are $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$

Fig. 2: Comparison of time complexity exponents between the classical case, the algorithms of [20] and our new results, with $\mathcal{O}\left(n\right)$ qubits only.

3.2 Quantum Algorithms for Unique k-xor

For Problems 4 and 3, we give algorithms in the QAQM model starting from $k = 3$. We improve over the previously known techniques for all k that are not multiples of 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*.*

3.3 k-xor with Classical Lists

In the QAQM setting, we give the first quantum speedups for Problem 1 for a general k. We prove Proposition 1. The details are given in Appendix D.

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 $\alpha_k \leq \frac{1}{2+\lfloor \log_2 k\rfloor}$.*

4 Introducing the k-Merging trees

In this section, we first present Wagner's algorithm [34] in two ways: first, as introduced in [34], 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 [15]. The description in [34] uses lists, but to emphasize the translation to a quantum algorithm, *we will start by considering Problem 2 instead*, with a random function $H : \{0,1\}^n \to \{0,1\}^n$.

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.

Example: 4*-xor.* The strategy for 4-xor is depicted on Figure 3. 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.

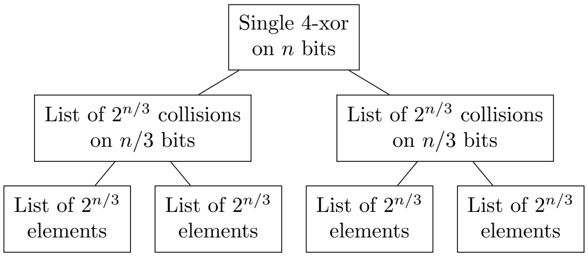

Single 4-xor on n bits
List of $2^{n/3}$ collisions on $n/3$ bits
List of $2^{n/3}$ collisions on $n/3$ bits
List of $2^{n/3}$ elements
List of $2^{n/3}$ elements
List of $2^{n/3}$ elements
List of $2^{n/3}$ elements

Fig. 3: 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 $z_1, \dots z_{k-2^\kappa}$ 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

To build a node of the tree, it suffices to have built its children; not necessarily all nodes of bigger depth. Wagner [34] 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 4, we highlight the difference between these two strategies, by considering the 4-xor tree of Figure 3. 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.

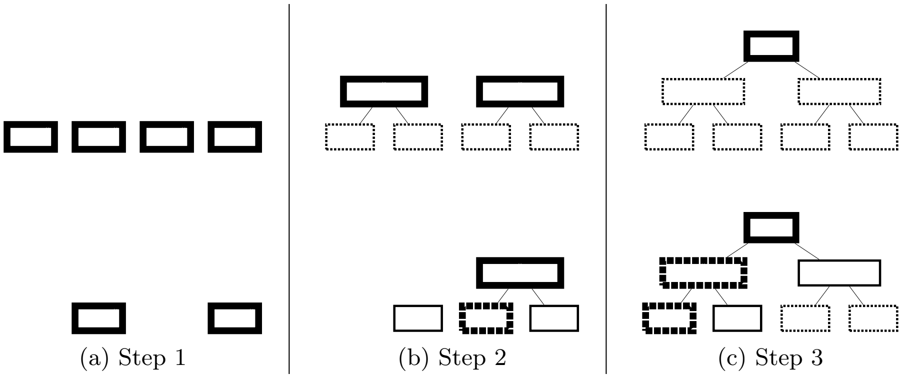

(a) Step 1 (b) Step 2 (c) Step 3

Fig. 4: Building the 4-xor tree of Figure 3 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 5.

1. We build and store the list L_0 of $2^{n/3}$ elements.

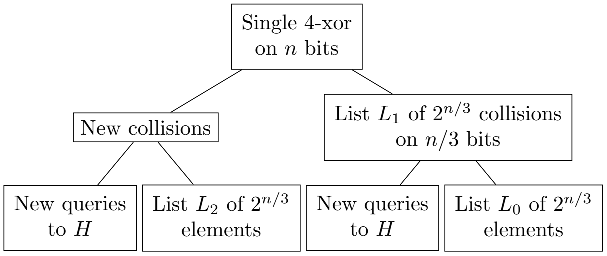

Single 4-xor on n bits
New collisions
List L_1 of $2^{n/3}$ collisions on $n/3$ bits
New queries to H
List L_2 of $2^{n/3}$ elements
New queries to H
List L_0 of $2^{n/3}$ elements

Fig. 5: Depth-first order in which to build the lists.

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).
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 [29] 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. This makes our analysis simpler, but we must pay a logarithmic overhead. The details can be found in Appendix A.1.

4.3 Limitations of the Extension to Quantum k-trees

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 (although this is proposed in [17]). 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 [20]. In their quantum walk approach, they 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 [28]. 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. In their low-qubits approach, 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

In the *depth-first* tree traversal for 4-xor of Figure 5, 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 5.

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 3 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 obtain a time complexity of $\widetilde{\mathcal{O}}\left(2^{n/4}\right)$, which is that of [20] for 4-xor. We don't use a quantum walk anymore, but the procedure still requires $\widetilde{\mathcal{O}}\left(2^{n/4}\right)$ QACM to hold the intermediate lists L_2 and L_1 during the final Grover search.

Moreover, the example of 3-xor shows that there exists *inherently quantum* merging strategies. In Algorithm 1, which also improves over [20], the corresponding "3-xor-tree" is of depth one. Classically, it does not yield a speedup over the collision exponent $\frac{1}{2}$.

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 6.

Algorithm 1 Quantum 3-xor Algorithm with QACM

```
Store a list $L_0$ of $2^{2n/7}$ elements;
Using Grover subroutines, build a list $L_1$ of $2^{n/7}$ elements with a $\frac{2n}{7}$-bit zero prefix;
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,
    • 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.
```

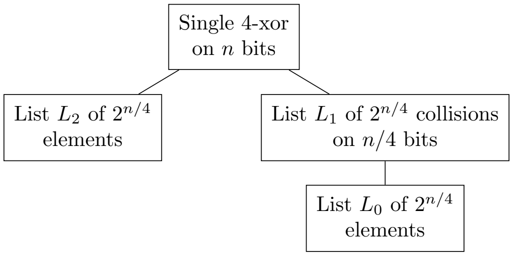

Single 4-xor on n bits
List L_2 of $2^{n/4}$ elements
List L_1 of $2^{n/4}$ collisions on $n/4$ bits
List L_0 of $2^{n/4}$ elements

Fig. 6: Tree of Figure 5 as an unbalanced quantum merging tree.

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 $T_0, \ldots T_{\ell-1}$, which are k_i-merging trees respectively, with the constraint $k_0 + \ldots + k_{\ell-1} = k - 1$.*

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 7.

- 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}

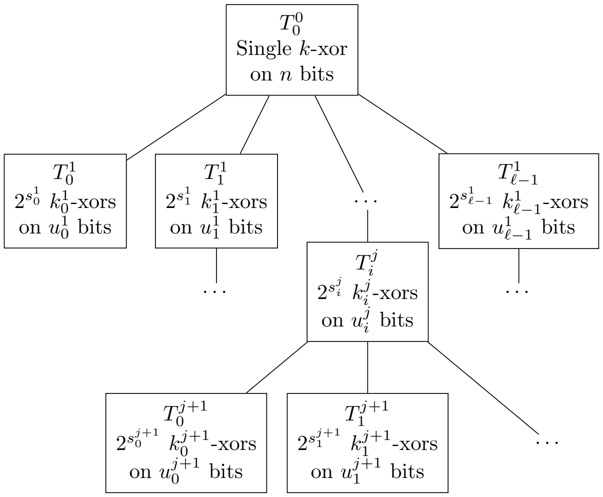


Fig. 7: k-merging tree

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 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, QACM 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 QACM: $c = s + \frac{1}{2}\left(u + u_0 - u_{\ell-1} - s_{\ell-1}\right)$*
- *Quantumly, low-qubits: $c = s + \frac{1}{2}\left(u - u_{\ell-1} - s_{\ell-1}\right) + \max\left(\frac{u_0}{2}, s_0, \dots, s_{\ell-1}\right)$*

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 quantum random 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 quantum random access, we can emulate a QACM by a sequential lookup of classically stored data. This was done in [16] in the case of quantum collision search (2-xor) and further used in [20] for low-qubit 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:

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

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, that we can solve with Mixed Integer Linear Programming (MILP). Given k, we can try different possible tree structures and find an optimal one.

Linear Program. We 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.

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 QACM and in the low-qubit setting. Our results are compared with the ones from [20] on Figures 1, 2 and Table 3 in Appendix.

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 [20] for many values of k. Furthermore, the quantum walk of [20] uses QAQM, while our method relies only on QACM. 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 $\lfloor \frac{1}{2} \left(k - \lfloor \frac{k}{2} \rfloor\right) \rfloor$-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 QACM and low-qubits setting (for $k > 5$) however it is not the only merging tree with such optimization.

QACM Setting. In the QACM case, each node that has a non-empty zero prefix is produced using Grover search. We note $\kappa = \lfloor \log_2(k) \rfloor$ and $\alpha_k = \frac{2^\kappa}{(1+\kappa)2^\kappa + k}$. 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 Setting. 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 [16] with $\alpha_2 = \frac{2}{5}$. For $k = 3$, we obtain the algorithm of [20] with $\alpha_3 = \frac{5}{14}$, showing that it remains optimal in our extended framework (contrary to 3-xor with QACM, 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 used by our algorithms, for an equal time, is always equal or better than the one from [20], in both settings. Notice that the low-qubits variants 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 QACM variant requires $\widetilde{\mathcal{O}}(2^{\alpha_k n})$ QACM (it is needed to store the leaf lists).

5.2 Optimality in the QACM 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 QACM 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:

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

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 Extended Merging Trees and Quantum Dissections

In this section, we extend merging trees to a much broader setting. We 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. All new algorithms in this section run in the QAQM model.

First we will show 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 $2^{n/k}$.

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

Our observation is that the *dissection* technique of [18, 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 $T_0, \ldots T_{\ell-1}$. 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 $T_0, \ldots T_{\ell-1}$ are built in time $t_0, \ldots t_{\ell-1}$ (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 $r_0 + \dots + r_{\ell-1} = r$. We can write the time complexity as:

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

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

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
...
$\ell - \mathbf{1}$ 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 [18, 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 [18, Section 3] to all intermediate domain sizes.

The quantum correspondence works in a very simple way: these ℓ nested loops become ℓ nested Grover searches. We search 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 $H : \{0,1\}^{n/4} \rightarrow \{0,1\}^n$. Classically, the best algorithm is Algorithm 2, from [33], in time $2^{n/2}$ and memory $2^{n/4}$. Quantumly, the best algorithm is in [7], in time $2^{0.3n}$ using $2^{0.2n}$ QAQM. Our method is Algorithm 3. It runs in quantum time $2^{0.3125n}$, smaller than a simple meet-in-the-middle, and QAQM $2^{0.25n}$. It is worse than [7] for Problems 4 and 3, but we will see in Section 7 that it can be used to attack the 4-encryption problem, contrary to [7].

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:

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

Algorithm 2 Classical 4-dissection

```
Query $H$ and store all the elements $H(x)$ in a list $L_0$
for each $u \in \{0,1\}^{0.25n}$ do
    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 $y \in L_0$).
    for each $z \in L_0$ do
        Find $t \in L_0$ such that $t \oplus z = u|*$.
        Find $x \oplus y \in L_1$ such that $x \oplus y \oplus z \oplus t$ gives a $0.5n$-bit zero prefix.
        If $x \oplus y \oplus z \oplus t$ is all-zero, then return this result.
    end for
end for
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.
```

6.2 Quantum Algorithms for Unique k-xor

In what follows, we solve together Problem 4 and 3 in the QAQM model, with the same time complexities. The reason why there is little difference between these problems is that, as long as quantum random-access is allowed (QACM or QAQM), it allows to simulate quantum oracle queries. It suffices to store the input data in QACM and replace an oracle query by a query to the whole memory. For $k \geq 4$, the cost of storing the whole domain of size $2^{n/k}$ is not dominant. For $k = 3$, there is a difference in the memory complexity. For completeness, the two procedures for $k = 3$ are given in Appendix C.

In the general case, we obtain Theorem 3. We do not improve the complexity of $2^{0.3n}$ for unique 4-xor obtained in [7], 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.

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.*

Without QAQM. Problem 4 becomes more difficult if QAQM is replaced by QACM. Indeed, assume that we are making a loop on a prefix of u bits, under which we build and store a list L of elements with u-prefix (before moving to other computations). It is crucial for our technique to be able to loop over this prefix with Grover search, in $2^{u/2}$ iterations. However, the list L written in each iteration is now in superposition as well, since it depends on u: it cannot be stored in classical memory. The solution would be to iterate classically on the prefix, in 2^u iterations. But then, we seem to loose the advantage over classical computations.

An algorithm for Problem 4 without QAQM can be obtained for $k = 3$ (and any multiple of 3) as follows: we store classically one of the lists and we do a Grover search on the product of the two others. The time complexity is always $\widetilde{\mathcal{O}}\left(2^{n/3}\right)$. We leave as an open problem to find QACM algorithms for unique k-xor (for any $k \geq 3$) with a factor less than 1/3 in the complexity exponent, or even to find algorithms in the low-qubits model.

7 Applications

In this section, we elaborate on various applications of our new algorithms. The common point of all the problems below is their k-list or *bicomposite* structure.

7.1 Improved Quantum time – memory Tradeoff for Subset-sums

Using the extended merging trees for Problem 3, we reach a better quantum time – memory product with respect to the current literature for low-density knapsacks, as was the case in [18] for classical algorithms.

Let $a_1, \dots a_n, t$ be randomly chosen integers on ℓ bits. We are looking for a subset of indices $I \subset \{1, \dots n\}$ 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 [23] takes time $\widetilde{\mathcal{O}}\left(2^{0.226n}\right)$, using as much QAQM.

A subset-sum problem can easily be translated to a k-sum problem with a single solution. Indeed, it suffices to separate the set $\{1, \dots n\}$ into k disjoint parts $J_1 \cup \dots \cup J_k$ and to start from the lists $L_1, \dots, L_k$, 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 (QAQM) 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), 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 $0.452n$.

7.2 New Quantum Algorithms for LPN and LWE

We consider the LPN problem in dimension n with constant error rate $0 \leq p < 1/2$. Given a certain number of samples of the form $(a, a \cdot s + e)$ where $a \in \{0,1\}^n$ is chosen uniformly at random and $e \in \{0,1\}$ is a Bernoulli noise: $e \sim Ber_p$ *i.e.* $P(e = 1) = p$. The LWE problem is the generalization from $\mathbb{F}_2$ to $\mathbb{F}_q$ for some prime q.

In [9], Blum, Kalai and Wasserman introduced an algorithm solving LPN in time $\mathcal{O}\left(2^{n/\log n}\right)$, using $2^{n/\log n}$ samples. Their idea is to combine samples: given $(a_1, a_1 \cdot s + e_1)$ and $(a_2, a_2 \cdot s + e_2)$ one can compute $(a_1 \oplus a_2, (a_1 \oplus a_2) \cdot s + e_1 + e_2)$. When summing k Bernoulli errors of correlation $\epsilon = 1 - 2p$, one obtains a Bernoulli error of correlation ϵ^k by the Piling-Up Lemma. Hence, the goal is to produce sufficiently many sums of a_i with almost all bits to zero, with sufficiently few a_i summed, so that one can obtain a bit of s from the samples gathered. The same principle applies to LWE, although we focus on LPN for simplicity.

In its original version, the BKW algorithm uses $2^{n/\log n}$ samples and memory. It starts from the list of samples and repeatedly finds partial collisions, cancelling $n/\log n$ bits in the a_i, until it produces a list of $2^{n/\log n}$ samples with a single nonzero bit. In [19], the authors find that there are many advantages of combining $c > 2$ samples at a time, that is, using a c-list algorithm in place of a simple 2-list merge operation. First of all, this reduces the memory used, which is crucial for practical implementations of the BKW algorithm. Second, this reduces the number of samples: we start from a smaller list. Finally, they are able to give the first quantum algorithm for solving LWE.

The c-sum-BKW algorithm is build upon the c-Sum-Problem as defined in [19, Definition 3.1]: *given a list L of N uniformly random b-bit strings, given $t \in \{0,1\}^n$ find at least N distinct c-tuples of elements of L that xor to t.*

They prove that, given an algorithm solving this problem in time $T_{c,N}$ and memory $M_{c,N}$ with overwhelming probability, for $b = \log c \frac{n(1+\epsilon)}{\log n}$ and $N \geq 2^{\frac{b + c \log c + 1}{c-1}}$, then their adapted BKW algorithm solves LPN in dimension n in time $T_{c,N}^{1+o(1)}$ and memory $M_{c,N}^{1+o(1)}$.

The authors study the solving of this c-sum problem via the Dissection method [18] and obtain new time-memory trade-offs. They also study a quantum version of this algorithm, hereby using a naive Grover search in the QACM model: we store L in QACM and perform a Grover search on all $c - 1$ tuples of L, for those who xor to an element of L. The memory used is N. As the parameters are tailored for N solutions in total, the quantum time complexity is $N^{c/2-1}$ for a single solution and $N^{c/2}$ for all of them. They leave as an open question (end of Section 1) whether a quantum k-list algorithm could be used in replacement.

New trade-offs. We are in a situation in which the input list is of size N_c and there are N_c solutions to recover. It is as if we were solving a c-xor problem on b bits with c lists of size $N = 2^{b/(c-1)}$ each, and wanted all the $2^{b/(c-1)}$ expected solutions. Furthermore, we limit the memory (QAQM) used to N. We simply solve the problem $\widetilde{\mathcal{O}}\left(2^{b/(c-1)}\right)$ times, as in the naive Grover case.

Table 1: Improvements on the quantum-BKW algorithm of [19] (see Table 1 in [19])

	Previous (naive + Grover)		This paper		
c	Memory	Time	Memory	Time	Time exponent
3	N_c	$N_c^{3/2}$	N_c	$N_c^{5/3}$	$5/3 = 1.667$
4	N_c	N_c^{2}	N_c	$N_c^{13/7}$	$13/7 = 1.857$
5	N_c	$N_c^{5/2}$	N_c	N_c^{2}	2
6	N_c	N_c^{3}	N_c	$N_c^{5/2}$	$5/2 = 2.5$
7	N_c	$N_c^{7/2}$	N_c	$N_c^{11/4}$	$11/4 = 2.75$
8	N_c	N_c^{4}	N_c	N_c^{3}	3

7.3 New Quantum Algorithms for the Multiple-encryption Problem

The multiple-encryption problem is an example of a *bicomposite* problem extensively studied in [18]. Consider a block cipher E_k with message space and key space of size n both. We consider the encryption by $E_{k_1} \circ \ldots \circ E_{k_r}$ with a sequence of independent keys $k_1, \ldots k_r$. Given r plaintext-ciphertext pairs (enough to discriminate the good sequence with high probability), we want to retrieve $k_1, \ldots k_r$. Classically, the best time complexity to date is essentially $2^{\lceil r/2 \rceil n}$ and the question is to obtain better time-memory trade-offs, as it is the case in [18]. We do not know of any r-list algorithm that wouldn't be applicable to r-encryption as well.

In [26], Kaplan proves that 2-encryption is (quantumly) equivalent to element distinctness[b]. However, already for $r = 4$, we remark that the 4-xor algorithm of [7] cannot be used to attack 4-encryption. Indeed, in the quantum optimization of [7], the size of the "intermediate value" that is guessed is not a multiple of n 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. 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.

[b] Kaplan [26] also gives an algorithm for 4-encryption, but we could not verify its time complexity.

We remark that all our r-xor algorithms (on nr bits) can be naturally converted to r-encryption: the size of the prefixes guessed is always a multiple of n, so we remain in a similar situation as [18], while this was not the case for quantum-walk based methods. For example, Algorithm 3 provides the best quantum time for 4-encryption that we know of, in quantum time $2^{1.25n}$ and QAQM 2^n to obtain the $4n$-bit key. Theorem 3 gives the best quantum time complexities for r-encryption for $r \geq 4$ and also shows an exponential decrease in the quantum time complexity with respect to 2-encryption.

7.4 Approximate k-list Problem

In [10], Both and May introduce and study the *approximate k-list problem.* It is 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 $\alpha = 0$). Its main application is solving the parity check problem: given an irreducible polynomial $P(X) \in \mathbb{F}_2[X]$ of degree n, find a multiple $Q(X)$ of $P(X)$ of a certain weight and degree. This is used in fast correlation attacks on stream ciphers. For this application, we can consider quantum oracle access (the lists actually contain polynomials of the form $X^a \mod P(X)$ for many choices of a).

The match-and-filter algorithm of [10, Section 3] consists in running a 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. With a quantum k-merging tree, we can always improve on this classical method in the QACM model. Let α_k be the k-xor optimal QACM time exponent as defined in Theorem 1. We cut the tree at its root: in time $\widetilde{\mathcal{O}}\left(2^{\alpha_k un}\right)$, we can obtain a tuple of lists $L_1, \dots L_t$ such that, given an n-bit element x, we can find $x_1 \in L_1, \dots x_t \in L_t$ such that $x \oplus x_1 \dots \oplus x_t$ has $(1 - 2\alpha_k)un$ bits to zero. Indeed, the Grover search at the root of the tree has also cost $\widetilde{\mathcal{O}}\left(2^{\alpha_k un}\right)$ since everything is balanced, so it eliminates $2\alpha_k un$ bits.

Hence, if we want to be able to eliminate un bits for some fraction $0 \leq u \leq 1$, we build all these lists in time $\widetilde{\mathcal{O}}\left(2^{\frac{\alpha_k}{(1-2\alpha_k)}un}\right)$.

Now we do a modified Grover search at the root: given any n-bit element x, the structure puts un bits to zero. There remains $(1-u)n$ (random) bits. We want the Hamming weight of the result to be less than a target $c_w n$. The proportion of $(1-u)n$-bit strings of Hamming weight less than $c_w n$ is approximately: $\binom{(1-u)n}{c_w n} / 2^{(1-u)n} \simeq 2^{(1-u)n(\mathrm{H}(c_w/(1-u))-1))}$ if $c \leq (1-u)$ and 1 otherwise, where H is the binary entropy function. Hence the number of Grover iterations in this last step is: $2^{\frac{1}{2}(1-u)n(1-\mathrm{H}_e(c_w/(1-u))))}$ where $\mathrm{H}_e(x) = 0$ if $x \geq 1$. It suffices to look for $0 \leq u \leq 1$ which optimizes the sum of the time complexities of the two steps:

$$2^{\frac{\alpha_k}{(1-2\alpha_k)}un} + 2^{\frac{1}{2}(1-u)n(1-\mathrm{H}_e(c_w/(1-u))))} .$$

We obtain the results of Table 2 by numerical optimization.

Table 2: Quantum speedup of the approximate k-list problem of [10], in the QACM model.

	$k = 2$			$k = 3$			$k = 4$	
c_w	$\log T/n$ (classical)	$\log T/n$ (quantum)	c_w	$\log T/n$ (classical)	$\log T/n$ (quantum)	c_w	$\log T/n$ (classical)	$\log T/n$ (quantum)
0	0.5000	0.3333	0	0.5000	0.2857	0	0.3333	0.2500
0.1	0.2920	0.1876	0.1	0.2769	0.1641	0.1	0.2040	0.1460
0.2	0.1692	0.1046	0.2	0.1590	0.0935	0.2	0.1238	0.0846
0.3	0.0814	0.0481	0.3	0.0778	0.0440	0.3	0.0630	0.0407
0.4	0.0232	0.0129	0.4	0.0221	0.0122	0.4	0.0195	0.0116

	$k = 8$			$k = 32$			$k = 1024$	
c_w	$\log T/n$ (classical)	$\log T/n$ (quantum)	c_w	$\log T/n$ (classical)	$\log T/n$ (quantum)	c_w	$\log T/n$ (classical)	$\log T/n$ (quantum)
0	0.2500	0.2000	0	0.1667	0.1429	0	0.1667	0.1429
0.1	0.1576	0.1200	0.1	0.1091	0.0889	0.1	0.1091	0.0889
0.2	0.0984	0.0714	0.2	0.0704	0.0548	0.2	0.0704	0.0548
0.3	0.0518	0.0355	0.3	0.0387	0.0284	0.3	0.0387	0.0284
0.4	0.0170	0.0106	0.4	0.0914	0.0091	0.4	0.0914	0.0091

8 Conclusion

Better Quantum k-xor Algorithms. In this paper, we proposed new algorithms improving the complexities from [20] for most values of k in both the QACM and low-qubits settings. We gave quantum algorithms for the k-xor problem with limited input size. 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. The merging trees we defined might 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. There are also many cryptographic applications for quantum k-list algorithms (*e.g.* lattice algorithms or decoding random linear codes [11]) that we did not cover yet.

Open Questions. 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 r-encryption encounters a limit $2^{0.3n}$. Whether an extended framework could allow to break this bound remains unknown to us. It would also be interesting to obtain better algorithms for Problem 4 (unique k-xor) without QAQM, or even in the low-qubits model.

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. Aaronson, S., Shi, Y.: Quantum lower bounds for the collision and the element distinctness problems. J. ACM 51(4), 595–605 (2004)
2. Ambainis, A.: Quantum Walk Algorithm for Element Distinctness. SIAM J. Comput. 37(1), 210–239 (2007)
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. Belovs, A., Spalek, R.: Adversary lower bound for the k-sum problem. In: Innovations in Theoretical Computer Science, ITCS 2013. pp. 323–328. ACM (2013)
6. Bennett, C.H., Bernstein, E., Brassard, G., Vazirani, U.V.: Strengths and weaknesses of quantum computing. SIAM J. Comput. 26(5), 1510–1523 (1997), https://doi.org/10.1137/S0097539796300933
7. 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)
8. 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)
9. Blum, A., Kalai, A., Wasserman, H.: Noise-tolerant learning, the parity problem, and the statistical query model. J. ACM 50(4), 506–519 (2003)
10. 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
11. 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)

12. Brassard, G., Hoyer, P., Mosca, M., Tapp, A.: Quantum amplitude amplification and estimation. Contemporary Mathematics 305, 53–74 (2002)
13. Brassard, G., Høyer, P., Tapp, A.: Quantum cryptanalysis of hash and claw-free functions. In: LATIN. LNCS, vol. 1380, pp. 163–169. Springer (1998)
14. Brassard, G., Høyer, P., Tapp, A.: Quantum algorithm for the collision problem. In: Encyclopedia of Algorithms, pp. 1662–1664 (2016)
15. 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)
16. 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)
17. Dinur, I.: An algorithmic framework for the generalized birthday problem. Cryptology ePrint Archive, Report 2018/575 (2018), `https://eprint.iacr.org/2018/575`
18. 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)
19. Esser, A., Heuer, F., Kübler, R., May, A., Sohler, C.: Dissection-bkw. In: CRYPTO (2). Lecture Notes in Computer Science, vol. 10992, pp. 638–666. Springer (2018)
20. 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)
21. 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`
22. 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`
23. 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)
24. 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)
25. Jaques, S., Schanck, J.M.: Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. IACR Cryptology ePrint Archive 2019, 103 (2019)
26. Kaplan, M.: Quantum attacks against iterated block ciphers. CoRR abs/1410.1434 (2014), `http://arxiv.org/abs/1410.1434`
27. Kuperberg, G.: Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem. In: TQC. LIPIcs, vol. 22, pp. 20–34. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
28. Magniez, F., Nayak, A., Roland, J., Santha, M.: Search via Quantum Walk. SIAM J. Comput. 40(1), 142–164 (2011)
29. Minder, L., Sinclair, A.: The extended k-tree algorithm. Journal of Cryptology 25(2), 349–382 (2012)
30. Newman, D.J., Shepp, L.: The double dixie cup problem. The American Mathematical Monthly 67(1), 58–61 (1960)
31. Nielsen, M.A., Chuang, I.: Quantum computation and quantum information (2002)

32. 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)
33. 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)
34. Wagner, D.A.: A Generalized Birthday Problem. In: Advances in Cryptology - CRYPTO 2002. LNCS, vol. 2442, pp. 288–303. Springer (2002)
35. Zhandry, M.: A Note on the Quantum Collision and Set Equality Problems. Quantum Info. Comput. 15(7-8), 557–567 (2015), http://dl.acm.org/citation.cfm?id=2871411.2871413

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

A node in a k-merging tree 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 [30] 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. Hence:

> When k is a constant, by multiplying all the lists sizes in the tree by a factor $2n$, the merging tree solutions will exist with probability $1 - negl(n)$.

This argument also ensures that each *classical* merging algorithm runs as predicted, *i.e.*, if a merging tree has predicted classical complexity $2^{\alpha n}$, then the practical complexity is $\widetilde{\mathcal{O}}\left(2^{\alpha n}\right)$. Furthermore, although we need to produce $2n$ more elements, the list sizes will actually remain the same as before: elements in a new list are produced on the fly and for each coupon (wanted prefix), we just need to keep one. For simplicity, we put this global factor aside and now turn ourselves towards the quantum algorithms, where other factors need to be taken into account.

A.2 Grover Search as a Building Block

Throughout the paper, in order to simplify our presentation, we assumed that Grover search with 2^t solutions in a search space of size 2^n ran in an exact number of $2^{(n-t)/2}$ iterations and with probability of success 1. If this was the case, the situation for quantum merging tree algorithms would be exactly the same as for classical algorithms, and the argument of Section A.1 would suffice.

However, Grover search differs from classical exhaustive search in two ways: 1. the number of iterations is actually $\left\lceil c2^{(n-t)/2} \right\rceil$ where $c = \frac{\pi}{4}$. Furthermore, the computations inside a Grover iterate need to be reversible, hence this constant c is often multiplied by 2 for uncomputations. 2. Grover's algorithm is a probabilistic procedure. In particular, the final measurement can yield an incorrect result. We now justify that, for all algorithms studied in this paper, if the predicted quantum time complexity of a merging tree is $2^{\alpha n}$, then the quantum time complexity for a constant success probability is $\widetilde{\mathcal{O}}\left(2^{\alpha n}\right)$.

Runtime of a Sequence of Grover Searches. In the merging trees defined in Section 4, nodes are built one after another, and each new node requires a sequence of Grover searches. Hence the whole quantum algorithm is a sequence of Grover searches, with access to intermediate lists stored in classical memory. An access to a stored list L_1 costs $\mathcal{O}\left(|L_1| n\right)$ quantum operations in the low-qubits model and $(\log |L_1|)^{\mathcal{O}(1)}$ in the QACM model. Each Grover iterations accesses at most k^2 (the maximal number of nodes in the tree) stored lists, whose size is exponential in n. Hence all low-qubits accesses can be assumed to cost $|L_1|$ and all QACM accesses can cost 1, with a global $poly(n)$ factor. If $L_1, L_2, \ldots$ are the successive lists built, the quantum time complexity becomes:

$$poly(n) \left(|L_1| \left\lceil c2^{(n-t_1)/2} \right\rceil + |L_2| \left\lceil c2^{(n-t_2)/2} \right\rceil + \ldots \right) \ .$$

We now justify that this quantum time is $\widetilde{\mathcal{O}}\left(2^{\alpha n}\right)$.

First of all, notice that in each of these instances of Grover's algorithm, the number of iterations is a single-exponential in n: $2^{\beta n}$ where β depends only on k (it comes from the tree structure). Hence $\left|\left\lceil c2^{(n-t_1)/2}\right\rceil - c2^{(n-t_1)/2}\right| \leq poly(n)$ where the polynomial depends only on k as well. Hence we can remove all $\lceil \rceil$ in the complexity computation, although the values are not integers anymore. We put c aside as a global constant factor. The sum obtained: $|L_1|2^{(n-t_1)/2} + |L_2|2^{(n-t_2)/2} \ldots$ is optimized to $2^{\alpha n} + 2^{\alpha n} + \ldots$ where $2^{\alpha n}$ is the promised time complexity of the quantum merging tree. There are at most k^2 nodes in the tree, hence we obtain $poly(n) \cdot k^2 2^{\alpha n} = \widetilde{\mathcal{O}}\left(2^{\alpha n}\right)$ as expected.

Error of a Sequence of Grover Searches. We now justify that the probability of success of the whole tree is $1 - negl(n)$. When Grover's algorithm runs with $\left\lceil c2^{\beta n}\right\rceil$ iterations, and $\beta > 0$ is a constant, the failure probability is smaller than $2^{-\beta n}$: it depends on how close the practical number of iterations, which is an integer, can be to the "ideal" $c2^{\beta n}$.

Crucially, when a failure occurs, we can easily detect it, as the result of each Grover search is immediately measured and stored in memory. Hence when Grover search fails with probability $2^{-\beta n}$, we need to multiply the number of searches by $1/(1 - 2^{-\beta n})$ (similarly as in Section A.1). We do that at each node, including the root. The increase in complexity remains negligible in n and the failure probability is negligible as well.

Runtime of Nested Grover Searches. In Section 6, we used nested instances of Grover's algorithm. 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. These nested instances come from the loops over guessed prefixes. The number of such loops can be at most k; inside a loop, there are at most k lists computed. By a similar argument as above, the multiplicative factor with respect to the predicted time complexity can be at most: $poly(n) \cdot k(2c)^k$.

Although the computation of new lists is now performed inside outer Grover searches, we can still take their error into account as before, by increasing the computational cost by a multiplicative factor negligible in n. However, the nested loops are instances of Grover search where the test function has some error probability (see [12]). The final probability of failure is the product of the probability of failure of each loop, which is negligible in n. As they are at most k nested loops, it remains negligible.

A.3 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 [34] 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.

A.4 Other Extensions of Merging Trees.

Inspired by the classical works that extended the original k-tree algorithm of Wagner [17,3,18,32,29], 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* [8] corresponds to taking a prefix of zeroes for the first leaf of a given node in the tree. This reduces memory usage at 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 [32]. 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 $T_0, \dots, T_{\ell-1}$, 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):

$$2^{u+v} + \dots + 2^{t-2u-v-\dots} \left(2^u + \dots\right)$$

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:

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

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:

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

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 [18, 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 (k-xor with many solutions and quantum oracle access).

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 8. 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.1). The total time is:

$$\underbrace{2^{\frac{8n}{45}}}_{\text{Iterations}} \left(\underbrace{2^{\frac{2n}{15}}}_{\substack{\text{Lookup of}\\ T_0^1}} + \underbrace{2^{\frac{n}{9}}}_{\substack{\text{Lookup}\\ \text{of } T_1^1}} + \underbrace{2^{\frac{2n}{15}}}_{\substack{\text{Lookup of}\\ T_2^1}} \right) = 2^{\frac{14}{45}n} \enspace .$$

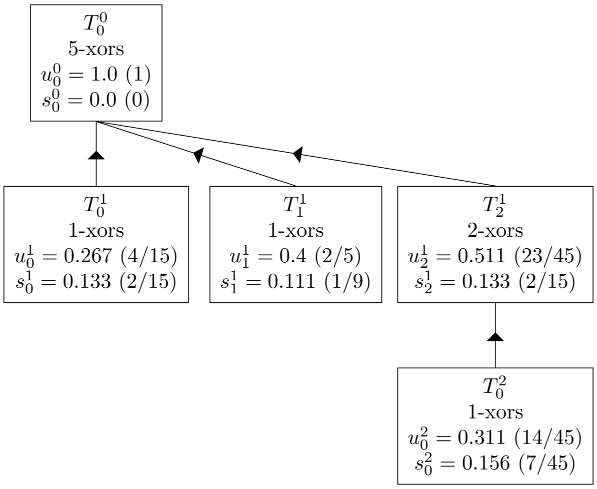


Fig. 8: Optimized Merging Tree for low-qubits 5-xor

B.2 Merging Tree Complexities for Problem 2

In Table 3, we give our results for solving Problem 2. 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 [20]. We highlight the improvements.

B.3 Proof of Optimality in the QACM 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 $\mathcal{T}_k$.

Table 3: Best time complexity exponents of k-xor optimized merging trees for k up to 12, and comparison with the results from [20].

	Classical		QACM	Low-qubits	Comparison QACM		Comparing Low-qubits	
k	Rounded	Fraction	Rounded	Rounded	This paper	[20]	This paper	[20]
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
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

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 $u_0 \leq \dots \leq u_{\ell-1}$.

Remark 1. For simplicity 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:

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

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

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

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 $u_{\ell-1}\alpha_{k_{\ell-1}} + 2 s_{\ell-1} \alpha_{k_{\ell-1}} = s_0$ 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 QACM 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:

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

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 $A(\alpha_{k_1}, \dots, \alpha_{k_{\ell-1}})$:

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

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

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

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:

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

in which we recognize $A(\alpha_{k_1}, \dots, \alpha_{k_{\ell-2}})$. 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 $A(\alpha_{k_1}, \dots, \alpha_{k_{\ell-2}})$, 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:

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

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 4 contains the constraints.

C Quantum Unique *k*-xor Algorithms

In this section, we give more details of the algorithms for problems 4 and 3. We first 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 4 and 5 differ only in that the latter allows access to a quantum oracle (which is the case for multiple-encryption), while the former supposes

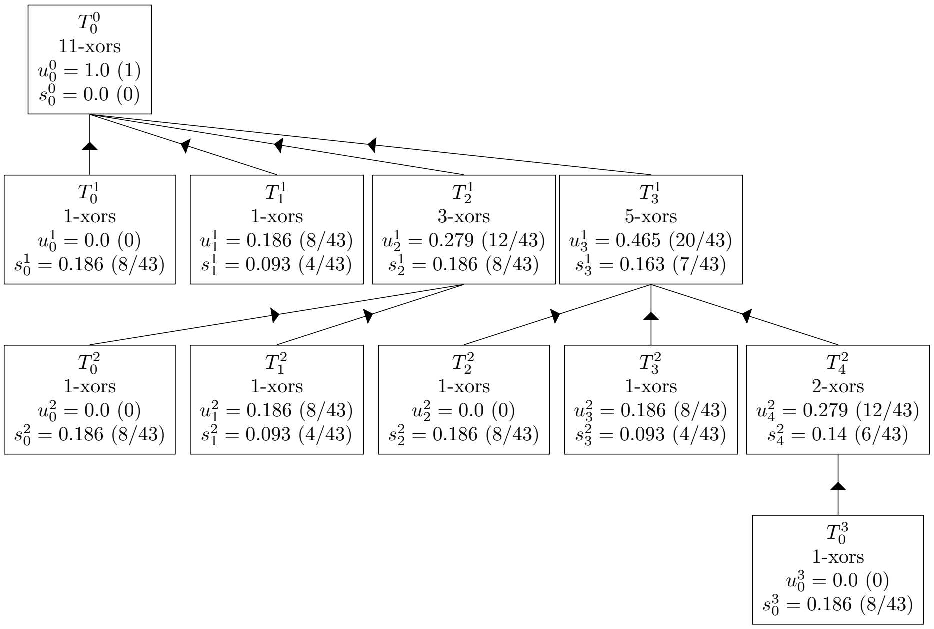


Fig. 9: 11-xor optimization of the tree $\mathcal{T}_{11}$

Table 4: 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	$s_0^0 = 0,\ u_0^0 = 1$ $u_3^1 = s_2^1 + u_2^1$ $u_2^1 = s_1^1 + u_1^1$ $u_1^1 = s_0^1 + u_0^1$	$c_0^0 = s_0^0 + \frac{1}{2}\left(u_0^0 + u_3^1 - u_0^1 - s_0^1\right)$

that the data is given classically. Both have a time complexity of $\mathcal{O}\left(2^{n/3}\right)$. Algorithm 5 uses only $2^{n/6}$ memory, while Algorithm 4 requires $2^{n/3}$ memory.

Algorithm 4 Quantum algorithm for the unique 3-xor problem, with QACM, without oracle access.

```
Store all the elements in a list $L_0$ (QACM-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
```

The time complexity of Algorithm 4 is:

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

Algorithm 5 Quantum algorithm for the unique 3-xor problem (or 3-encryption), with QAQM 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 5 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 6 is $2^{n/10}\left(2^{n/5} + 2^{n/5}\right) = 2^{3n/10}$. It uses a memory $2^{0.2n}$.

Finding the Best Time-Memory Product. In general, to reach the time $\widetilde{\mathcal{O}}\left(2^{0.3n}\right)$, our algorithms for unique k-xor require an amount of $2^{0.2n}$ QAQM, so they do

Algorithm 6 Quantum algorithm for the unique 5-xor problem, 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 QAQM
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 7 Translation of algorithm 6 to the 5-encryption problem.

```
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
```

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 QAQM, and may require active hardware (see [22] or [25]). Finding the best one 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 5. They are difficult to describe, especially since the best algorithms with this respect are not the best algorithms in time. For example for $k = 10$, we have an algorithm running in time $\widetilde{\mathcal{O}}\left(2^{0.35n}\right)$ using $2^{0.1n}$ memory. At $k = 12$, with time $\widetilde{\mathcal{O}}\left(2^{n/3}\right)$ and memory $2^{n/12}$, the product is minimal among the instances studied. We do not know whether it is a minimum over all k.

Table 5: Our quantum time and QAQM 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.

	Our best time		Corresponding QAQM		Best t.-m. product	
k	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

D 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, QAQM access seems necessary to achieve a time speedup against Wagner's algorithm, so as before, we present results only in the QAQM 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 quantum random-access. We can also assume that they are sorted. 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 [20] 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 8) of quantum time complexity: $2^{n/7}\left(2^{n/7}+2^{n/7}\right)=2^{2n/7}$. In general, we obtain the results of Figure 10 and Table 6.

Algorithm 8 Quantum algorithm for the 4-list problem with QAQM

```
Store the input lists $L_0, L_1, L_2, L_4$ of size $2^{2n/7}$ of $n$-bit elements in memory
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
```

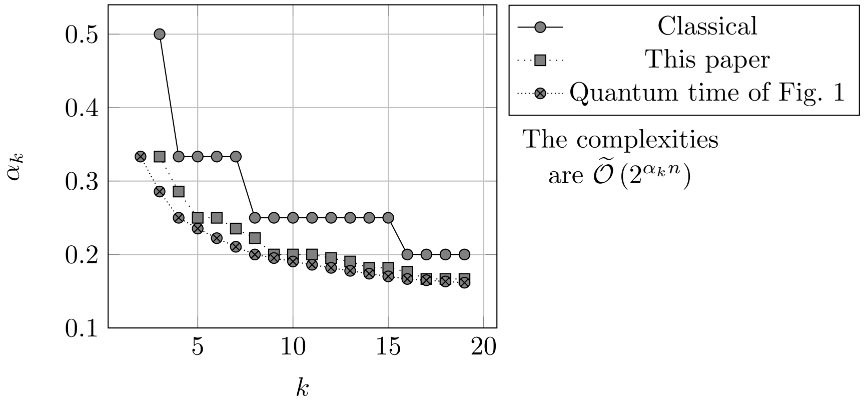


Fig. 10: Quantum time complexities of the k-xor problem with classical input data, compared with the k-xor with a quantum oracle, and with the classical time.

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 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.* □

In Table 6, we give the exponents obtained for Problem 1 (k-xor with classical inputs).

Table 6: Complexity of the k-xor problem with classical inputs, compared with quantum oracle access, as $\widetilde{\mathcal{O}}\left(2^{\alpha_k n}\right)$.

	Classical α_k		Without oracle		With oracle	
k	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