

Building Quantum Circuits with Qarton

André Schrottenloher

Univ Rennes, Inria, CNRS, IRISA, Rennes, France

`firstname.lastname@inria.fr`

Abstract. In order to estimate accurately the impact of large-scale computing devices, one requires a precise resource estimate of quantum algorithms for cryptographic problems. As a start, these algorithms can be described as quantum circuits. One then estimates the number of gates, qubits, and circuit depth that they require, as a first step towards physical resource estimates.

In order to simplify this task, different quantum programming languages have been used, targeting different circuit sizes and abstraction levels. Quantum cryptanalysis is indeed a domain in which one may consider both medium-scale circuits with fine-grained optimizations (such as optimizations of Shor’s algorithm) and very large-scale circuits with well-defined building blocks (such as exhaustive key search on block ciphers using Grover’s algorithm).

This paper presents Qarton, a python library to represent logical quantum circuits, focused on quantum cryptanalysis. It allows to study very large-scale circuits like Grover searches, to simulate classical or “almost” classical circuits such as the arithmetic building blocks of Shor’s algorithm, and to analyze trade-offs between qubit and gate count when switching between different implementations of arithmetic operations. Qarton is equipped with a library of around 300 circuits and examples from different areas of quantum symmetric and asymmetric cryptanalysis.

Keywords: Quantum algorithms, Quantum circuits, Quantum cryptanalysis

1 Introduction

In cryptography, *cryptanalysis* is necessary to quantify the security offered by a cryptosystem. A continuous, fine-grained study of potential attack algorithms eventually builds a foundation of trust for cryptographic tools.

Despite being still under development, quantum computers have already had a major impact on cryptography. This impact is commonly summarized into two important applications:

- Shor’s algorithm [Sho94] solves the Abelian hidden subgroup problem, and in particular, factors integers and finds discrete logarithms in polynomial time, breaking RSA and cryptography based on Abelian groups, such as elliptic curve groups and finite fields;
- Grover’s algorithm [Gro96] speeds up generic black-box search problems from a classical time T into a quantum time $\mathcal{O}(\sqrt{T})$ (quadratic speedup), reducing the generic level of security offered by most primitives, including symmetric ciphers.

There exist many more examples of cryptographic problems admitting quantum speedups using dedicated algorithms. Some of them are *less* than quadratic; for example, quantum collision search [BHT98] and all its applications in symmetric cryptography, lattice sieving algorithms [CL21, BCSS23], and generic decoding algorithms [KT17], which are important problems for the security of post-quantum asymmetric cryptosystems. Some of them

are *more* than quadratic, like the offline-Simon algorithm applied to certain block cipher key-recovery attacks [BSS22], and, in group action-based cryptography, Kuperberg’s algorithm [Kup13], which is directly related to the security of post-quantum cryptosystems based on group actions [CLM⁺18, Pei20].

Quantum Programming for Cryptanalysis. In this context, breaking down complex quantum algorithms at the logical quantum circuit level, and analyzing very precisely the required resources (gates, qubits, depth) serves two purposes, depending on the algorithm that we are studying.

An algorithm like Shor’s [Sho94] and its numerous variants ([GRLR⁺23, Lit23, Gid25] to cite only a few) contains a relatively moderate amount of resources: at most thousands of qubits and the order of tens of millions [BZG⁺26] to billions [Gid25] of gates. While this is out of the scope of currently available machines, it is useful to explore trade-offs between gate count and qubits at the logical level, and overall to make the circuit more compact.

Meanwhile, Grover’s exhaustive search on the block cipher AES-128 [JNRV20] is expected to cost in the range of 2^{80} Toffoli gates. This massive requirement is unlikely to be realized by medium-term quantum computing devices, and indeed, it was used in the NIST post-quantum standardization process as the defining quantum security of the “NIST 1” level [NIS16]. More generally, many quantum cryptanalysis studies – similarly as classical cryptanalysis – work in *fundamentally infeasible* regimes. The goal here is to perform a fine-grained study of security levels, compare algorithms, and perhaps lay out the components for more effective attacks. Therefore, even though the algorithm is not expected to be implemented, it is desirable to obtain a reliable and precise cost estimate, at least down to a factor 2. As an example, dozens of works [ASAM18, GLRS16, JBK⁺25, LPS20, ZWS⁺20, SF24, LPZW23, LXX⁺23, JNRV20, HS22] have refined the quantum resource estimates of the AES block cipher, which can not only be applied to Grover’s exhaustive search, but to essentially any attack algorithm on AES-based cryptographic designs. These works improved over the first implementation by a factor around 2^5 in gate count. This has slightly moved the definition of the NIST 1 level, but the attack remains infeasible.

In addition to cryptanalytic applications, similar arithmetic building blocks intervene in quantum algorithms which, while not immediately applicable to standardized cryptography, could be used as proofs of concept of verifiable quantum advantage [KMRVK25, KSG⁺25].

Implementation Examples. Over the years, different quantum programming frameworks to obtain detailed cryptanalytic resource estimates, targeting the different scenarios mentioned above. Here are a few examples.

- Jaques et al. [JNRV20] (quantum key search), Häner et al. [HJN⁺20] (Shor’s algorithm on elliptic curves), Bonnetain and Jaques [BJ22] (offline-Simon algorithm on symmetric primitives) used the Q# language [SGT⁺18].
- Chevnard et al. [CFS24] (quantum information set decoding), Chevnard et al. [CFS25] (quantum factoring), Edme et al. [EFS25] (solving Boolean quadratic systems) and Dupont-Latapie and Perret [DLP26] (algebra) used the Qiskit Python framework [JTK⁺24].
- Khattar et al. [KSG⁺25], in their work on using DQI circuits for quantum advantage (which contain heavy arithmetic components) used the Qualtran Python framework [HKY⁺24].

This list is not exhaustive, and there exist many other programming languages or frameworks like Qrisp [SBZ⁺24], ProjectQ [SHT18] or Quipper [GLR⁺13] which have been used for resource estimates, not only in cryptanalysis.

Sometimes, other studies used ad hoc resource estimates, like Bernstein et al. [BLMP19] (evaluating the CSIDH group action). However, in many cases, concrete resource estimates are simply performed by hand, using simplified or asymptotic formulas.

The fragmentation of these implementations, which use different frameworks and are rarely interoperable, makes it more difficult to build upon existing results: the authors of new cryptanalysis studies often have to re-implement the most simple building blocks of their algorithms, like addition circuits.

Requirements for Quantum Cryptanalysis. A simple solution to the interoperability problem would be to re-implement the algorithms in an existing framework, like Qiskit [JTK⁺24], Qrisp [SBZ⁺24], Qualtran [HKY⁺24], Quipper [GLR⁺13], etc. In order to be applicable to quantum cryptanalysis circuits, this framework should also allow the following:

- **Scale** to large circuit sizes (tens of thousands of qubits, millions of gates) and even to “infeasible” circuits, e.g. with 2^{80} gates, arising in quantum algorithms. While we do not expect a framework to represent an arbitrary sequence of 2^{80} gates, we know that most quantum cryptanalysis circuits, and also many arithmetic circuits, are obtained by repeating sub-routines. For example, Grover’s search is obtained by iterating a sub-circuit (the Grover iterate); we should represent it indeed as an *iterated* operation and focus on the complexity of this sub-circuit.
- Allow **fine-grained** resource estimates, where we not only count the basic gates, but also different sub-circuits. As an example, since the cost of the AES block cipher is dominated by the cost of its S-Box layer, it would make sense to simplify the cost estimate by counting S-Box circuits only (while still having a representation of the full circuit).
- Allow to **simulate** large-scale classical and “almost” classical circuits. By “almost” classical, we mean a circuit that, starting from any basis state, constructs a sparse state vector (a quantum state supported by a small number of basis states). A typical example would be a purely classical circuit made from reversible Boolean logic gates (X, CX, CCX). On input a single basis state, it returns a single basis state, and at any point, the superposition is supported by a single basis state. The extension of classical circuits with measurement-based uncomputation (notably when uncomputing AND gates, like in the Gidney adder [Gid18]) has been described in [BZG⁺26] as *kickmix circuits*, which, during simulation, only produce a single basis state and a phase. More generally, it has been studied in [JH22] that leveraging the sparsity of quantum states can make simulations easier, especially for instances of quantum cryptanalysis like Shor’s algorithm.
- Allow to switch between **manual** and **automatic** memory management. Quantum computations being reversible, an important part of quantum circuit design is the management of *ancilla qubits* (temporary work qubits which are reverted to 0). Advanced frameworks like Qualtran [HKY⁺24] and Qrisp [SBZ⁺24] manage the ancillas automatically: the user does not have to allocate ancillas when a sub-circuit requires them, which is otherwise a tedious task. However, it should be also possible for the user to easily circumvent the default allocation strategy. For example, when computing the AES block cipher, the S-Box layer contains 16 parallel S-Boxes, each requiring a few ancilla qubits. If the automatic allocator reuses these ancillas for each successive circuit, the total depth will be multiplied by 16. Instead, one should be able to allocate separate ancillas for each circuit.
- Interface easily with **classical** functions. Eventually, we need to devise quantum circuits that operate on complex algebraic objects, like elliptic curve points. It should

be easy to define custom data types that can be easily interfaced with the quantum circuit framework, and allow to test quickly that the implementation is correct.

- Switch easily between **different implementations** of the same component. Inside a complex circuit like elliptic curve addition, many different arithmetic operations are computed: modular inversion, modular multiplication, etc. Each of them can be implemented by different circuits. It should be possible to try different implementation strategies, e.g., to switch between the low-space CDKM adder [CDKM04] and the low-gate Gidney adder [Gid18], without having to rewrite the code, and furthermore, these choices should be made *implicitly* whenever this makes sense.

To the best of our knowledge, none of the aforementioned frameworks handle natively all these requirements. Qiskit [JTK⁺24] does not allow to represent circuits with *iterated operations* having an arbitrary number of iterations, and does not automatically manage memory. Qualtran [HKY⁺24] does it, but switching between sub-circuit implementations is done *explicitly*: when declaring a circuit, indicating which circuit should be used as adder, etc.

Arguably, one could build over an existing framework to handle all these needs, but we eventually decided that it would be less challenging to start from a new, minimal design, partially inspired by existing approaches, but mostly driven by practical requirements.

Qarton. Qarton¹ is a Python library to represent medium- and large-scale logical quantum circuits, to perform resource estimates on them, and to analyze trade-offs resulting from different implementations.

The Qarton “core” (`qarton.circuit`) is a relatively small package with about 11000 lines of code. The rest of the library is organized into circuit packages, which define an extensive range of about 300 quantum circuits: unsigned and signed arithmetic, modular arithmetic, linear algebra, elliptic curves over prime fields, quantum amplitude amplification [BHMT02], etc.

Qarton uses a simple syntax, similar to Qiskit [JTK⁺24], where quantum circuits are constructed explicitly, by forming a sequence of sub-circuits and basic quantum gates. In addition, we provide built-in support for *iterated operations*, which allow to represent easily large-scale iterated circuits like Grover searches – even though such circuits cannot be simulated. Memory is managed automatically by default, but can also be managed manually, allowing to perform trade-offs between the number of qubits and the depth. The inputs and outputs of circuits are defined using *data types*, with automatic type-checking (similarly to Qualtran [HKY⁺24]). Simulation of circuits is performed using a built-in Python simulator supporting sparse state vectors (similarly to [JH22]). To ensure that it remains efficient for large-scale circuits, sub-circuits with classical behavior can be replaced by *dummy* classical functions when applicable, which are also useful for automatic testing. Finally, we provide support to switch between different implementations (e.g., of an adder circuit) using the same code, by declaring *backend specifiers* when creating a circuit, allowing automatic time-space trade-offs.

The design philosophy of Qarton is that **Circuit** objects should be *explicit*: the number of qubits, their positions in the circuit, and the positions at which sub-circuits are applied, are all determined when constructing the circuit (“instantiating” it as an object). Therefore, Qarton is not “compiled”, and resource estimates consist simply in traversing the circuits. The resulting code is intended to be easy to read, unambiguous and somewhat easy to convert into another framework if needed.

¹“Carton” means “cardboard” in French. This was just an attempt at being funny.

Code. This document refers to version **1.0.2** of the library, which can be found at:

<https://gitlab.inria.fr/capsule/qarton>

The documentation (installation instructions, API and tutorials) can be found at:

<https://capsule.gitlabpages.inria.fr/qarton/>

Organization. This note is not intended as a complete documentation of **Qarton**, for which we refer to the online documentation and API. Instead, our goal is to give some context on quantum circuit programming, definitions relevant to the library, descriptions of basic and more advanced functionalities and small examples of code.

Section 2 gives some preliminaries of quantum computing (logical quantum circuits) and defines the basic gates of **Qarton**. **Section 3** explains how to define a quantum circuit, starting from a simple example, and defining more complex functionalities like iterated sub-circuits, measurements and debugging. **Section 4** explains how memory is managed by **Qarton**. It also explains the lifetime of *registers* during circuit construction.

The next sections detail advanced usage of the library. First, in **Section 5**, we explain how to define a new datatype. Then, in **Section 6** we explain how *backends* work, how to define a *backend handler* capable of switching between several circuit implementations, how to specify the backends of a circuit, and which syntax to use to ensure that backends are used correctly.

2 Preliminaries

This section gives useful preliminaries of quantum circuits, with some explanations on how these concepts are mapped into **Qarton**. We refer to [NC10] for a more detailed exposition of the quantum circuit model.

In **Qarton**, we stay purely at the *logical* quantum circuit level. The considerations at the *physical* level, which is constrained by the hardware and the error-correcting codes used, are very different. Optimizing circuits at the logical level is still useful for two reasons. For algorithms like Shor’s, which might eventually be realized physically, we will always start by logical circuit estimates before mapping the circuits to a physical implementation. For other algorithms like Grover key search on AES, which are not expected to run in practice, a detailed logical implementation allows to make meaningful estimates of security levels, and precise comparisons between algorithms.

2.1 Quantum States

A *qubit* is an elementary quantum system with two basis states $|0\rangle, |1\rangle$, forming a basis of a Hilbert space isomorphic to $\mathbb{C}^2$. Such a state can be written as a linear combination:

$$\alpha |0\rangle + \beta |1\rangle \tag{1}$$

where $|\alpha|^2 + |\beta|^2 = 1$. The orthogonal vectors $|0\rangle, |1\rangle$ form the *computational basis*. The values α, β are complex *amplitudes*, whose argument is named the *phase*. **Qarton** works in this basis. More generally, the state of an n -qubit quantum system is of the form:

$$\sum_{i=0}^{2^n-1} \alpha_i |i\rangle \tag{2}$$

where i should be – by default – considered as an n -bit string representing the state of the individual qubits. Quantum states are always normalized: $\sum_i |\alpha_i|^2 = 1$. Note that in terms of notation, $|a\rangle |b\rangle$ is equivalent to $|ab\rangle$ when a and b are two bits or bit-strings.

Informally, we say that a superposition “contains” a basis state, or that a basis state “belongs” to it, if its amplitude in the superposition is non-zero. In simulations, we represent amplitudes with a fixed precision, so there is a threshold below which a basis state is not considered anymore as part of the support, and simulation results may differ from theoretical estimates. Nevertheless, the precision issue is minor compared to the *number of amplitudes* that a faithful simulation needs to represent.

A *classical state* can be seen as a special case of a quantum state, in which the superposition is restricted to a single basis state. This allows to embed classical computations into quantum computations. The circuits constructed with **Qarton** contain both *quantum bits* and *classical bits*; the latter keep the same value in any basis vector of the superposition. This does not really make a difference to the simulator.

2.2 Quantum Computations

A *computation* in the quantum circuit model is a sequence of operations which turn a quantum state into another quantum state. An operation can be either a unitary operator of $\mathbb{C}^{2^n}$, a *measurement* of a qubit, or a *reset* of a classical bit.

Measurements. A *measurement* is a non-unitary operation that turns some qubits into classical bits and transforms the quantum state according to the following rule. Consider a quantum state rewritten as:

$$\sum_{i=0}^{2^n-1} \alpha_i |i\rangle |\psi_i\rangle \quad (3)$$

where the $|\psi_i\rangle$ themselves are normalized quantum states. Measuring the bits of i then yields the state $|i\rangle |\psi_i\rangle$ with probability $|\alpha_i|^2$. The bits of i now have a fixed value.

A *reset* is an operation that can only be used on previously measured qubits. It erases the value contained in these qubits, and replaces it by 0. In **Qarton**, we merge these two operations into a single *measure-set-reset* (MSR). MSR is applied on a pair (q, c) where q is (the index of) a qubit and c (the index of) a classical bit.

- q is measured, and becomes a classical bit b ;
- c is set to the value b ;
- q is reset, and becomes 0.

MSR is the only non-unitary basic operation that we need. One should note that in a **Qarton** circuit, classical and quantum bits will always remain at separate positions, so the MSR is the operation that *transfers* a measured value towards a classical bit.

Unitaries. Between two measurements, a quantum state can only evolve through unitary operators. We decompose such operators into elementary *quantum gates*, and most often, sub-circuits. An important property of a unitary operator is that it is *invertible*: the inverse of U is denoted $U^\dagger$ and often called the *uncomputation* of U . When U is decomposed as a sequence of gates, the uncomputation of U consists in the inverse sequence of gate inverses. It will have therefore the same gate count and use the same number of qubits. **Qarton** provides automatic circuit inversion using the method `.inverse()`. If the circuit contains measurements, it may still have a well-defined inverse, but in that case the inverse should be provided by the user, as it cannot automatically be inferred from the circuit definition. When the inverse circuit is user-defined, we require it to have *exactly the reverse memory layout* of the original circuit. This restriction allows `.inverse()` to operate correctly for all invertible circuits.

2.3 Quantum Gates

Qarton provides the following basic quantum gates.

Classical Logic Gates. A *classical* gate maps any basis state to another basis state. Since we can only implement unitary operators (except when operating on classical bits), we are limited to classical reversible gates:

- X gate (also known as the NOT gate, from classical logic): maps $|b\rangle$ to $|\bar{b}\rangle$ where b is a bit.
- CX gate (also known as CNOT): maps $|a\rangle|b\rangle$ to $|a\rangle|a \oplus b\rangle$ where a, b are bits.
- CCX gate (also known as Toffoli gate, or CCNOT): maps $|a\rangle|b\rangle|c\rangle$ to $|a\rangle|b\rangle|a \oplus bc\rangle$ where a, b, c are bits.

The X+CX+CCX gate set is universal for reversible logic. This means that any permutation operating on n -bit strings can be decomposed as a circuit of X, CX and CCX gates. However, for any $n \geq 4$, odd permutations will require at least one *ancilla* (or auxiliary) bit [SPMH03]. For convenience, we also include:

- SWAP gate: maps $|a\rangle|b\rangle$ to $|b\rangle|a\rangle$.

In the logical circuit model, which is the setting of Qarton, gates can be applied to any qubits without restrictions on their placement. The SWAP gate amounts to a relabeling of the qubits. In our constructions it is explicit, but costs nothing.

Quantum Phase Gates. While the gates above modify the basis states (they perform a bit flip), the following ones modify the phase. They also operate independently on basis states.

- Z gate: $|a\rangle \rightarrow (-1)^a |a\rangle$.
- CZ gate: $|a\rangle|b\rangle \rightarrow (-1)^{ab} |a\rangle|b\rangle$.
- CCZ gate: $|a\rangle|b\rangle|c\rangle \rightarrow (-1)^{abc} |a\rangle|b\rangle|c\rangle$.
- T gate: $|a\rangle \rightarrow (\exp(i\pi/4))^a |a\rangle$. That is, a controlled phase multiplication by $\exp(i\pi/4)$.
- S gate: $S = T^2$.
- $T^\dagger$ and $S^\dagger$ gates.
- General phase gate P_ϕ : $|a\rangle \rightarrow \exp(i\phi)^a |a\rangle$.
- General controlled-phase gate C_ϕ : $|ab\rangle \rightarrow (\exp(i\phi))^{ab} |ab\rangle$.

The general controlled-phase is a convenient gate in some circuits, but it is not strictly necessary to achieve universality. In addition, Qarton provides a “global phase” operation which will apply a global phase to the quantum state. Quantum states that differ only by a global phase are indistinguishable. We only use this to correct a global phase factor that would arise when simulating a given circuit.

Other Gates. The remaining gates are the Hadamard gate, which modifies the amplitudes in the superposition, and the MSR gate, which is also treated as a “quantum gate” by Qarton.

- H (Hadamard) gate: $|x\rangle \rightarrow \frac{1}{\sqrt{2}} |0\rangle + \frac{(-1)^x}{\sqrt{2}} |1\rangle$
- MSR gate: measures a qubit, resets it, sets a classical bit to the measured output.

```

3 from qarton.circuit import Circuit, UInt, UIntType
4
5
6 class TTKAdder(Circuit):
7     def __init__(self, n: int) -> None:
8         super().__init__()
9         qr_x = self.add_input_output(UIntType(n))
10        qr_y = self.add_input_output(UIntType(n))
11        for i in range(1, n):
12            self.cx(qr_x[i], qr_y[i])
13        for i in reversed(range(1, n - 1)):
14            self.cx(qr_x[i], qr_x[i + 1])
15        for i in range(n - 1):
16            self.ccx(qr_y[i], qr_x[i], qr_x[i + 1])
17        for i in reversed(range(1, n)):
18            self.cx(qr_x[i], qr_y[i])
19            self.ccx(qr_y[i - 1], qr_x[i - 1], qr_x[i])
20        for i in range(1, n - 1):
21            self.cx(qr_x[i], qr_x[i + 1])
22        for i in range(n):
23            self.cx(qr_x[i], qr_y[i])
24
25
26 if __name__ == "__main__":
27     qc = TTKAdder(5)
28     assert qc.simulate((UInt(4), UInt(5))) == (UInt(4), UInt(9))

```

Figure 1: Modular adder construction from [TTK10], and simulation. [\[Link to code\]](#)

2.4 Choosing the Computational Basis

So far our presentation has been centered on the *computational basis*, but we could also represent quantum states in a different basis, or switch the basis depending on the state. Qarton does not do that: in the definition of circuit, their inputs, their outputs, and their simulation, we keep to the computational basis. This seems actually more adapted to quantum cryptanalysis, where states in the computational basis are quite sparse, as noticed in [JH22]: the support of the superposition is on a small subset of the computational basis, which might not be true if we move to another basis.

3 Programming a Quantum Circuit with Qarton

This section briefly explains the principles of programming quantum circuits with Qarton, taking a few examples.

3.1 Circuit from a Gate Sequence

As a first example, we program a generic addition circuit. This is a construction from [TTK10] which adds modularly and does not return a carry.

On this code we can detail the basic principles of Qarton.

- A circuit is defined by inheriting from the class `Circuit`. Instances of (subclasses of) `Circuit` are automatically memoized² (unless the subclass is decorated with

²In previous versions this was the other way around: circuits had to be decorated with `@memoize` to be memoized. In the current version this decorator still exists, but does nothing.

`@no_memoize`). This means that calling again the same circuit constructor with the same arguments will give us the *same python object*.

- The circuit is constructed by allocating registers and defining the sequence of operations. All of this happens inside the `__init__` constructor of the class. Each circuit is a fixed-size object, so calling `TTKAdder(4)` or `TTKAdder(5)` will create new circuit objects with 4-bit and 5-bit inputs respectively.
- The construction of the circuit is deterministic from the arguments passed to `__init__`, ensuring that memoization makes sense.
- The circuit declares several *input* and *output* registers. An input register is declared using the method `add_input` (which returns a register), while an output register is declared using `add_output` (which takes a register as argument). The method `add_input_output` does both and indicates a *preserved* register, which has the same type and the same position in the input and output of the circuit. Adding inputs and outputs to the circuit populates its input and output *templates* and *signatures* (defined below). An important part of circuit signatures is to identify such *preserved* registers. The case of `TTKAdder` is simple, because it operates on two registers which are preserved: the inputs (x, y) and the outputs $(x, x + y)$ have the same types and stay in the same place.
- Registers have types. The `TTKAdder` circuit constructed here operates on two unsigned integers (`UInt` objects) which are declared with `UIntType(n)` (unsigned integers of size n).
- In this circuit, we are only applying basic gates, which are available through methods of the `Circuit` class (similarly to Qiskit). Basic gates apply on individual qubits.

Signature and Template. The (input / output) *Template* of a circuit gives information on its *functionality*: the types of the inputs and if they are classical or quantum.

The *Signature* of a circuit (obtained by `qc.sig`) gives a complete information on its memory layout in input and output: total number of qubits it operates on, positions of the inputs and outputs, their datatypes, position of classical bits and dirty ancillas (workspace qubits which can take arbitrary values and need to be reinitialized). Two circuits may have the same template but different signatures: the registers may be positioned differently, they may use a different number of ancillas, etc.

After constructing the circuit with an appropriate size, we can simulate it on an input matching its input template (here a pair of integers, as requested by the types), and it will return an output matching its output template (here a pair of integers again).

Constraint on Preserved Registers. In order to simplify circuit construction and signatures, we impose the additional constraint that *preserved registers should always come first*. That way, we ensure that they will always keep the *same position* in the template. This constraint is not difficult to enforce when constructing circuits, but should be kept in mind when using them. As an example, the `EuclideanDivider` from `qarton.arithmetic` computes a Euclidean division $x = qy + r$, mapping (y, x) to (y, q, r) . Because y (the divider) is unmodified, it is the *first* input of the circuit.

3.2 Registers

A *register* in Qarton is an object that is created:

- When creating an input;

- When allocating an ancilla, a dirty ancilla or a classical register;
- When appending a sub-circuit.

Internally, the register holds some *absolute positions* of bits in the circuit (the first bit starts at 0 and we increment the position when appending new bits) and a corresponding *datatype*. Thus a register is a volatile object: it just contains the information “the bits at these positions can be interpreted as an object of this datatype”.

Remark 1. Since a quantum circuit is operating on a quantum state, by “bits at these positions”, we mean the bits in any basis state of the superposition. In other words, the quantum state that we operate on is always *homogeneous* regarding typing; we do not consider cases in which the type interpretation would be conditioned on some qubits (if this was the case, then it should be defined by the type itself).

Remark 2. We are saying “bit” here, as this includes qubits and classical bits. Classical bits may be positioned arbitrarily in the circuit – but classical and quantum bits will always remain separate.

An important point, related to automatic memory management, is that the absolute positions of bits *always remain unknown* to the user – yet they exist explicitly and are determined at each step of circuit construction. Obtaining these positions may only be useful for testing or debugging.

A register being a volatile object, some operations can *destroy* it. This mostly happens when applying certain sub-circuits, which transform their inputs, “reshuffle” the qubits and scramble the original datatypes. From the user’s perspective, these operations consume registers and return new ones. See [Subsection 4.3](#) for more details on the destruction of registers.

3.3 Qarton Operations

Circuits are constructed in **Qarton** by applying operations to registers, either through basic gates, or by appending sub-circuits with **append**. We provide 4 types of operations which cover most of the use cases we have found in quantum cryptanalysis:

- A “basic” operation applies a basic gate (including measure / reset) or a quantum sub-circuit at specified positions in the circuit.
- An “iterated” operation applies a fixed number of iterates of a sub-circuit. The sub-circuit needs to be an *in-place* circuit (which preserves its input registers); for example, an iterate of Grover search.
- A “controlled” operation applies a sub-circuit if, at simulation time, some classical bits are 1. Note that this is not a *quantumly* controlled operation (the current name is a bit unfortunate).
- A “conditional” operation is a more general operation that applies a sub-circuit defined at simulation time. The sub-circuit is defined by a function of some classical values, which are taken from (currently classical) registers in the circuit.

While controlled operations are essential in measurement-based uncomputations, iterated operations – alongside memoized circuits – are useful to efficiently construct large circuits in **Qarton**.

```

3 from qarton.arithmetic import ControlledTTKCarryAdder
4 from qarton.binary_operations import Shift
5 from qarton.circuit import *
6
7
8 class _MultiplierIterate(InPlaceCircuit[tuple[UInt, UInt, UInt]]):
9     def __init__(self, bit_size_x: int, bit_size_y: int) -> None:
10         super().__init__()
11         x_reg = self.add_input_output(UIntType(bit_size_x))
12         y_reg = self.add_input_output(UIntType(bit_size_y))
13         out_reg = self.add_input_output(UIntType(bit_size_y + bit_size_x))
14
15         self.append(
16             ControlledTTKCarryAdder(len(y_reg)),
17             qc_bool_reg(x_reg[0]),
18             y_reg,
19             qc_reg_cast(out_reg[0:bit_size_y], datatype=UIntType(bit_size_y)),
20             qc_bool_reg(out_reg[bit_size_y]),
21         )
22         self.append(Shift(len(x_reg), offset=-1), x_reg)
23         self.append(Shift(len(out_reg), offset=-1), out_reg)
24
25
26 @dummify
27 class Multiplier(Circuit[tuple[UInt, UInt], tuple[UInt, UInt, UInt]]):
28     def __init__(self, bit_size_x: int, bit_size_y: int) -> None:
29         super().__init__()
30         x_reg = self.add_input_output(UIntType(bit_size_x))
31         y_reg = self.add_input_output(UIntType(bit_size_y))
32         out_reg = self.add_anc_output(UIntType(bit_size_y + bit_size_x))
33
34         self.append_iterated(
35             _MultiplierIterate(bit_size_x, bit_size_y),
36             x_reg,
37             y_reg,
38             out_reg,
39             iterations=bit_size_x,
40         )
41         self.append(Shift(len(out_reg), offset=bit_size_x), out_reg)
42

```

Figure 2: Multiplier circuit. [\[Link to code\]](#) Note that this code is correct, but does not handle properly *backend switching*, which is covered in another section.

Example: Multiplier Circuit. In Figure 2, we show an example of an iterated sub-circuit, and give some more details on circuit programming.

A multiplier of two integers can be built from a controlled-adder circuit by performing a series of additions of the second integer, controlled on the bits of the first integer, and bit-shifts. In Figure 2, we define a `Multiplier` circuit that follows this strategy.

This multiplier has two input registers `x_reg` and `y_reg`, which are unsigned integers. These registers, being defined with `add_input_output`, are *preserved*. An output register `out_reg` is allocated and set as output by `add_anc_output`. The circuit iterates `len(x_reg)` times a circuit `_MultiplierIterate`, and appends a `Shift` sub-circuit to correct the current shift of the output register.

The method `append` and its variants (like `append_iterated`) take as input a sub-circuit and any number of registers corresponding to the inputs of the circuit, and return registers corresponding to the outputs. Since the only circuits we are appending here are in-place, the input registers are not destroyed, and there is no need to capture the outputs (although they are still returned).

In `_MultiplierIterate`, the controlled adder is applied on the following registers: • the first bit of `x_reg` as control; • `y_reg`; • the first `len(y_reg)` bits of `out_reg`, considered as an unsigned integer of size `len(y_reg)`, and finally the next bit of `out_reg`, used to XOR the carry. We use some built-in functions from `qarton.circuit` to manipulate bits and registers:

- We transform the bit `x_reg[0]` into a register object by calling `qc_bool_reg`. The register has type `BoolType`. We often need to do this with control qubits.
- Though we can operate on registers with list syntax, like `out_reg[0:bit_size_y]`, the obtained register will always lose its type and have a default `BitVectorType`. We use `qc_reg_cast` to cast explicitly the register with the correct datatype.

Thanks to this structure, constructing a multiplier over n bits, which contains $\mathcal{O}(n^2)$ gates in total, only takes time $\mathcal{O}(n)$. Most of the time is spent constructing the adder and `Shift` circuits (both have a linear number of gates which are applied sequentially). Note that `Shift` only contains SWAP gates.

Remark 3 (Cost estimates). The allowed operations in `Qarton` are intended to offer as much flexibility as possible, while still allowing reliable cost estimates. For an iterated operation, depth and gate counts are obtained by multiplying the depth and gate counts of the operation. For controlled operations, we always count the sub-circuit by default (that way, we will at most over-estimate the total operation counts of running the circuit) – this can be modified. For conditional operations, we count the costs of a default sub-circuit instantiation, and assume that all possible instantiations have similar costs.

Example: Grover Search. In Figure 3, we give a simple implementation of Grover’s algorithm, where we search for the all-one bit-string among the 2^n possible n -bit strings. The circuit `Flip` is a test that flips the phase if the bit-string is all-one, and we also use it in the diffusion transform. The Grover iterate is defined as a temporary sub-circuit, which we memoize via `.freeze()`. It is then fixed and can be used in `append_iterated`. The number of iterations is set so that the probability of success is high. The simulator runs for $n = 4$, but wouldn’t terminate for a large n . Yet we can still construct the circuit for $n = 128$ and obtain its gate count.

3.4 Simulation

The datatypes of input and output registers handle conversions to and from Python objects (which should be of type `QartonObject`). For example, this modular adder circuit takes unsigned integers and returns unsigned integers:

```

3  from math import pi, sqrt
4
5  from qarton.binary_operations import Flip
6  from qarton.circuit import BitVector, Circuit, InPlaceCircuit, ResourceReporter
7
8
9  class GroverSearch(Circuit[None, BitVector]):
10     def __init__(self, n: int) -> None:
11         super().__init__()
12         search_reg = self.add_anc_output(n)
13         self.h_reg(search_reg)
14
15         # define the iterate
16         grover_iterate = InPlaceCircuit()
17         _tmp = grover_iterate.add_input_output(n)
18         # flip the phase if all bits are 1
19         grover_iterate.append(Flip(n), _tmp)
20         # apply the Grover diffusion transform: flip
21         # the phase if all bits are 0
22         grover_iterate.h_reg(_tmp)
23         grover_iterate.append(Flip(n, negate=True), _tmp)
24         grover_iterate.flip_global_phase()
25         grover_iterate.h_reg(_tmp)
26         grover_iterate.freeze()
27
28         # append the Grover iterate
29         it = int(sqrt(pi / 4 * (1 << n)))
30         self.append_iterated(grover_iterate, search_reg, iterations=it)
31
32
33  if __name__ == "__main__":
34     qc = GroverSearch(4)
35     print(qc.simulate_state(None))
36     # returns a state where BitVector(1,1,1,1) has a large amplitude
37
38     qc = GroverSearch(128)

```

Figure 3: Grover search circuit. [\[Link to code\]](#) Note that this code is sufficient at this point, but does not handle properly *backend switching*, which is covered in another section.

```

3 from qarton.circuit import *
4 from qarton.arithmetic import TTKAdder
5
6 qc = TTKAdder(5)
7 assert qc.simulate((UInt(2), UInt(3))) == (UInt(2), UInt(5))

```

Here `UInt` is simply a redefinition of Python `int`, which is equivalent to a Python `int` at runtime. It was added simply to distinguish integers used as Qarton objects from those used in the rest of the code.

To simulate large-scale circuits, it is useful to replace classical sub-circuits by *dummy classical functions*. Such functions also serve as a single point of truth for the expected functionality of the circuit, allowing to test it automatically. Figure 4 demonstrates this on the Multiplier circuit.

```

26 @dummify
27 class Multiplier(Circuit[tuple[UInt, UInt], tuple[UInt, UInt, UInt]]):
28     def __init__(self, bit_size_x: int, bit_size_y: int) -> None:
29         super().__init__()
30         x_reg = self.add_input_output(UIntType(bit_size_x))
31         y_reg = self.add_input_output(UIntType(bit_size_y))
32         out_reg = self.add_anc_output(UIntType(bit_size_y + bit_size_x))
33
34         self.append_iterated(
35             _MultiplierIterate(bit_size_x, bit_size_y),
36             x_reg,
37             y_reg,
38             out_reg,
39             iterations=bit_size_x,
40         )
41         self.append(Shift(len(out_reg), offset=bit_size_x), out_reg)
42
43     def dummy_classical_function(
44         self, args: tuple[UInt, UInt]
45     ) -> tuple[UInt, UInt, UInt]:
46         x, y = args
47         return x, y, UInt(x * y)
48
49     def dummy_classical_function_inverse(
50         self, args: tuple[UInt, UInt, UInt]
51     ) -> tuple[UInt, UInt]:
52         x, y, _ = args
53         return x, y
54
55     def validate_output(self, args: tuple[UInt, UInt, UInt]) -> bool:
56         x, y, z = args
57         return z == x * y

```

Figure 4: Multiplier circuit with dummy functions. [\[Link to code\]](#)

In general, the methods to be implemented are:

- A “dummy classical function” that should match exactly this circuit’s `simulate` routine. In particular, it takes as many inputs as the circuit needs, and returns as many outputs, with types matching the circuit’s template.

- A “dummy classical function inverse”, which is the dummy classical function for this circuit’s inverse. Indeed, most circuit inverses are created automatically using `.inverse()`, so we need to specify what the inverse function does as well.
- An “input validation” routine and an “output validation” routine (explained below).

When applying a quantum circuit, we make assumptions on its input, and derive assumptions on its output. For example, we assume that the input bit-string contains valid type elements at the positions indicated by input registers, and that the other bits are 0. This assumption is implicit. But we also often make other assumptions, which need to be explicit.

In the `Multiplier` example, the `Multiplier` circuit does not make any assumptions on its inputs. However, when applying the inverse circuit, we are applying it on three registers `x_reg`, `y_reg`, `out_reg`, and we are effectively assuming that `out_reg` contains the product of `x_reg` and `y_reg`. This assumption ensures that the circuit runs correctly and uncomputes `out_reg` (leaving all its bits to 0).

`Qarton` does *not* provide a generic way to check assertions on input and output registers at circuit construction time. Since we are not proving that the code is correct, we check this by simulating the circuits.

In the `Multiplier` example in Figure 4, a valid output of the circuit is a triple `x,y,z` where `z` is the product of `x` and `y`. When simulating the dummy inverse circuit, we will run `validate_output`, and will raise an error if the assertion is unsatisfied.

3.5 Resource Estimation

One of the objectives of `Qarton` is to perform resource estimation at a large scale. By *resources*, we mean the following quantities:

- Space: number of quantum and classical bits, where we are mostly interested in qubits.
- Depth: the *depth* of a circuit is the maximal number of gates applied in sequence, assuming that gates operating on independent qubits can be applied in parallel. More generally, we count the depth relative to a *depth context*, which gives a certain depth to a certain subset of sub-circuits or basic gates.
- Gate and sub-circuit counts: we count the number of sub-circuits according to the circuit *names* (which are instance-dependent), or to *class names* (which are class-dependent).

All resource estimates can be accessed by a `ResourceReporter` object³. Figure 5 shows an example.

3.6 Measurements and Classical Functions

As we have mentioned above, `Qarton` “circuits” can contain measurements. A measurement is an operation performed on a qubit *and* a classical bit, where the value of the classical bit is set to the measured value, and the qubit is always reinitialized to 0. This is the operation `msr` of the class `Circuit`.

We can also use:

- The method `measure` that will automatically allocate a classical bit and perform `msr` (for a single bit);

³Previous versions of `Qarton` used a separate class `AndFanoutResourceReporter`, but the current version does everything with `ResourceReporter`, and the current `AndFanoutResourceReporter` is simply an alias for `ResourceReporter`. New code should use `ResourceReporter` only.

```

3 from qarton.circuit import ResourceReporter
4 from qarton.modular_arithmetic import ModularAdder
5
6 if __name__ == "__main__":
7     qc = ModularAdder(17)
8     r = ResourceReporter(qc)
9
10    print(r.nbr_bits(), r.nbr_qubits())
11    print(r.ccx_count())
12    print(r.ccx_count_log2())
13    print(r.sub_circuits_by_name())

```

Figure 5: Resource estimates. [\[Link to code\]](#)

- The function `qc_measure` that takes a quantum register, measures and resets it (the register cannot be used afterwards), and returns a newly-allocated classical register with the same datatype;
- The method `measure_reg` of the class `Circuit` which is a shorthand for calling `qc_measure`.

After measuring classical values, we might have some classical post-processing to do before the next operations of the circuit. `Qarton` does not define a specific “classical operation”, but it has a “dummy circuit” class that we use for this. The class `DummyCircuit` defines circuits that are never “real”, and only defined by a dummy classical function and / or their metrics. A classical function (class `ClassicalFunction`) is a subclass of `DummyCircuit` which purposefully has only classical inputs and outputs, and no quantum operations: it has depth 0 and 0 quantum gates. Because the circuit is purely classical, its function does not need to be reversible. It can erase or drop its input.

```

3 from qarton.circuit import *
4
5
6 class HadamardThenMeasureThenXOR(Circuit[None, QartonBool]):
7     def __init__(self) -> None:
8         super().__init__()
9         xreg = self.add_anc(5)
10        self.h_reg(xreg)
11        creg = self.measure_reg(xreg) # destroys xreg
12        out = self.classical_function( # does not destroy creg
13            lambda x: QartonBool(sum(x) % 2), # type: ignore
14            creg,
15            output_type=BoolType(),
16        )
17        self.add_output(out)
18
19
20 if __name__ == "__main__":
21     qc = HadamardThenMeasureThenXOR()
22     print(qc.simulate(None)) # prints 1 or 0

```

Figure 6: Measurements. [\[Link to code\]](#)

In [Figure 6](#) we define a quantum circuit with a classical function inside. The circuit

performs a Hadamard transform on a zero-register, then measures to obtain a random bit-string. It then computes the XOR of the bit-string and outputs this.

This classical computation (internally performed using a dummy circuit) is defined by the method `classical_function`. It takes as input a callable, and multiple registers in which we will read the inputs of this function. The function should output a single `QartonObject`, which is written into a new register; we pass a `DataType` to indicate how to type this output. In our case, we are computing the XOR of all input bits, and outputting a single `QartonBool` (boolean value).

The callable, and whether it can be truly applied on the data read from the registers, cannot be checked at circuit construction time, so we need to simulate to check that it works.

3.7 Debugging

A large part of `Qarton` code debugging happens at *circuit construction time*. When trying to instantiate a quantum circuit, `Qarton` will notice mismatched datatypes, classical bits used instead of qubits, or ancilla qubits which haven't been marked as restored.

Debugging can also be done at *simulation time*, simply by trying to simulate the circuit on random inputs (this is what we do in the library of circuits). But it is often useful to check some assertions *in the middle* of the circuit while it's being simulated. Internally, this is done by appending special `Circuit` instances which have no operations, but call some functions during simulation (a behavior that is similar to the `ClassicalFunction` circuits above, even though they are two separate interfaces).

The user interface for these debugging tools is as follows:

- `self.print(r)`: prints the contents of the register `r`, interpreted according to its datatype, when the circuit is simulated. Note that if there are multiple basis states traversed during simulation, then the printing occurs for all of them.
- `self.test_anc(r)`: checks that the register `r` is 0, and raises an exception if it's not the case.
- `self.generic_assert()`: checks an assertion (defined by a `Callable`, like a lambda) on the contents of its registers. The callable interface here is similar to `classical_function`, except that there is no output (the output of the callable is a Boolean, and we check at runtime that it's True). The callable's input type is `QartonCircuitType`, so if there are multiple registers, the input should be a tuple.

4 Memory Management

This section explains briefly how memory management is performed in `Qarton`.

As mentioned at the end of [Section 3](#), programming with quantum circuits often involves applying circuits on registers which satisfy certain assumptions. Because these assumptions are unknown to `Qarton`, one can easily construct impossible circuits. Most of the time, this involves a value which is not uncomputed correctly. Consider the code of [Figure 7](#).

This circuit admits an integer input and returns no output, meaning that it should always erase its input: this would not be reversible, so we know it's impossible. After declaring the input register, we call `assert_anc`, which indicates that the register passed in input is now “ancilla”, i.e., zero. This method is commonly used to indicate that manually-managed qubits have been cleared (other languages, like Quipper [\[GLR⁺13\]](#), have similar assertions). Because all qubits are now zero, there is no need to return an output register, and the class `BadCircuit` can be instantiated without error.

```

3 from qarton.circuit import *
4
5
6 class BadCircuit(Circuit[UInt, None]):
7     def __init__(self, r: int) -> None:
8         super().__init__()
9         x_reg = self.add_input(UIntType(r))
10        self.assert_anc(x_reg)

```

Figure 7: A “bad” quantum circuit showing an impossible operation.

Errors only occur when we try to simulate it. With `UInt(0)` as input, we don’t get any error, because all bits are indeed 0 at the end. With any other input, we get an `InvalidNonzeroBits` exception, because after simulating the circuit, we have obtained non-zero bits in positions where the signature doesn’t expect them.

4.1 Manual or Automatic Memory Management

Memory management in Qarton is *semi-automatic* and *declarative*, in the following sense:

- We allow the user to *explicitly* declare ancilla registers and manage them. The user can declare ancilla registers using `add_anc` (allocate a new ancilla register) or `get_anc` (declare a new ancilla register using existing or new ancillas – this is almost always preferable to `add_anc`). After performing operations on the registers, the qubits are considered to be non-zero. If the ancillas have actually been cleared out, the user needs to call `assert_anc` to indicate this (and then, `release_anc` can be used to “free” the register, as indicated below).
- We also secretly manage ancillas to be used silently by sub-circuits. When calling `append`, it is possible to pass a register of ancillas to be used by the sub-circuit; otherwise, the circuit’s memory manager will fetch and / or allocate the necessary ancilla qubits.

Switching between automatic and manual ancilla management is useful to perform depth-width tradeoffs. Indeed, the default manager’s behavior is sequential: it reuses ancillas from one operation to the next. If we want to apply sub-circuits in parallel, we can instead allocate manually ancilla registers and pass them to `append`.

This semi-automatic memory management is implemented by a `MemoryManager` class, which handles the actual allocation of registers and their absolute positioning in the circuit. Since we assume no locality constraints, Qarton comes with a default `MemoryManager` which uses a greedy allocation strategy.

The manager partitions the circuit’s memory into:

- Classical bits, which are handled separately;
- Ancilla qubits (which are currently known to be 0);
- Used qubits (which are currently believed to be non-zero); including *dirty* ancilla qubits (which need to be restored at the end of the computation).

Operations like basic gates will transform ancilla into used bits. An input register starts as used bits. In addition, ancilla bits are split into *protected* and *unprotected* ancilla. *Protected* ancilla are those which are currently available to the user, as they belong to explicitly named registers. For example, the operation: `r = self.get_anc(UIntType(4))` initializes an ancilla register with type `UIntType(4)` in the circuit. By *protected*, we mean

that we allow the manager to use these ancillas in sub-circuits, but only if they are restored afterwards – they cannot be silently consumed and moved to the used bit pool, as this would break the user’s logic. Conversely, ancillas that were automatically created are named *unprotected*. They are not known to the user and cannot be used as inputs of gates or operations; they can only be used automatically when appending sub-circuits which require ancillas. This may move them to the used pool, if they become part of newly created registers.

Moving ancilla qubits between the protected and unprotected pools is possible, and allows some memory optimizations. This is done with the methods:

- `r = self.get_anc(UIntType(4))` to move qubits from the unprotected pool to the protected pool, returning an ancilla register with the requested type (it will also extend the pool if necessary);
- `self.release_anc(r)` to move qubits from the protected pool to the unprotected pool, destroying the register `r` in the process.

It is almost always preferable to use `get_anc` to obtain ancilla qubits and `release_anc` to discard qubits of which we know that they are zero again, unless we have immediately a use for them.

Finally, we can also manage classical bits. Since classical bits do not need to be restored to 0, there is only a notion of *protected* and *unprotected* bits. We can use the methods:

- `r = self.get_classical(UIntType(4))` to move classical bits from the unprotected pool to the protected one, returning a classical register with the requested type;
- `self.release_classical(r)` to move classical bits from the protected pool to the unprotected one, destroying the register `r` in the process. Classical bits do not need to be 0 to be released.

Input and output registers of circuits can be classical, and this information is part of the circuit’s template and signature (to declare a classical input, we use a Boolean optional argument of `add_input`).

4.2 Features and Limitations

The main limitation of **Qarton** memory management is that it does not handle locality constraints. Indeed, since we do not know absolute positions, we must assume that applying gates or sub-circuits at any positions costs the same. **Qarton** works only at the logical level, and locality constraints in actual physical implementations should be handled separately.

In addition, the memory management depends crucially on the methods `assert_anc` or `release_anc` being used correctly (we can use the method `test_anc` instead to debug this at simulation time).

A useful feature of **Qarton** circuits is that the memory allocation is abstract: we can create circuits of arbitrary sizes, as shown in Figure 8.

```

3 from qarton.circuit import *
4
5 large_circuit = Circuit()
6 large_circuit.add_input_output(UIntType(1 << 32))

```

Figure 8: A large quantum circuit. This code will not make your computer crash.

Creating an adder circuit following Figure 1 of size n will take a time roughly linear in n , mostly because we add the operations one by one. In Figure 8 we just initialized a register without any operations; this takes constant time.

This feature is useful to scale up Qarton circuits as long as they depend on sub-circuits which are iterated or repeated many times. For example, we could put a thousand adder circuits in parallel, like in Figure 9. The construction remains efficient, though the simulation will become infeasible eventually.

```

3 from qarton.circuit import *
4 from qarton.arithmetic import TTKAdder
5
6 parallel_adders = Circuit()
7 for _ in range(1000):
8     x = parallel_adders.add_input_output(UIntType(1000))
9     y = parallel_adders.add_input_output(UIntType(1000))
10    parallel_adders.append(TTKAdder(1000), x, y)

```

Figure 9: Many adders in parallel.

4.3 Uncomputation, Destroying and Remapping

Due to their reversibility, circuits often create *garbage* values which must be *uncomputed* while the results are kept along. Some frameworks like Qrisp [SBZ⁺24] allow for some kind of automatic uncomputation. This has also been studied in [PBSV21, HH25, VGDV24]. We did not implement such a feature in Qarton. Instead, we always perform uncomputing by appending inverse circuits. While the inverse circuit can be automatically created (using `.inverse()`), it is often up to the user to decide in which order the values will be computed and uncomputed, and to apply the inverse operation at the appropriate time⁴.

Example: Sign. In Figure 10, we define a circuit that computes the absolute value of a signed integer (copied from the library). Due to reversibility, the circuit must return *two* values: an unsigned integer (`UInt`), and a bit of sign (`QartonBool`, which is an alias for a Python `int`). The input register is destroyed.

We use this circuit to compute only the sign, out of place. The circuit that we define takes as input a signed integer `SInt`, which should be preserved (so we use `add_input_output`). We start by appending the absolute value circuit. Doing so returns the two registers `a`, `s` (absolute value and sign) and *destroys* the register `r`.

We copy the sign `s` into a new register `sp`, and add it as an output. Then, we need to uncompute. We append the inverse of the absolute value circuit to `a`, `s`. Doing so returns a register `rr`.

A subtlety is that `rr` is a *new* register, with no guarantee that it would be in the same positions as the original register `r`, even though it will be the case here, since a circuit and its inverse have reversed memory layouts. The workaround is to finish the circuit construction by calling `.remap`. This function is aware of the internal memory layout. It takes the current registers which should form the circuit’s output (two in our case) and applies SWAP gates if necessary to move the qubits at their expected positions. In our case, the qubits of `r` must be in the same positions in input and output, so `remap` will satisfy this constraint.

Preserving registers is a desirable property for quantum circuits, as this simplifies higher-level circuits, makes the code easier to read (we do not have to redefine registers

⁴That said, we also provide in the library a generic construction `ComputeUncompute` for simple cases, where we just need to compute forwards, keep some output of the circuit, and erase all the rest.

```

3  from qarton.arithmetic import qc_cincr_uint
4  from qarton.binary_operations import qc_copy, qc_fanout
5  from qarton.circuit import *
6  from qarton.signed_arithmetic import *
7
8
9  class Abs(Circuit[SInt, tuple[UInt, QartonBool]]):
10     def __init__(self, width: int) -> None:
11         super().__init__()
12         xreg = self.add_input(SIntType(width))
13         x, s = xreg[:width], qc_bool_reg(xreg[width])
14         qc_fanout(s, x)
15         x = qc_reg_cast(x, UIntType(width))
16         qc_cincr_uint(s, x)
17         self.add_output(x)
18         self.add_output(s)
19
20
21  qc = Circuit()
22  r = qc.add_input_output(SIntType(4))
23  a, s = qc.append(Abs(4), r)
24  sp = qc_copy(s)
25  qc.add_output(sp)
26  (rr,) = qc.append(Abs(4).inverse(), a, s)
27  qc.remap(rr, sp)
28  qc.freeze()
29
30  # Note that this is very much *not* what we should do in
31  # practice, since a register of type SIntType already has a field "sgn"
32  # that gives the sign bit. So this is equivalent:
33
34  qc = Circuit()
35  r = qc.add_input_output(SIntType(4))
36  s = r.a["sgn"]
37  sp = qc_copy(s)
38  qc.add_output(sp)
39  qc.freeze()
40

```

Figure 10: Circuit computing the absolute value of a signed integer, and how to use it to compute only the sign out-of-place. [\[Link to code\]](#)

after each operation), allows iterated operations to be defined, and reduces the possibility of errors. Whenever this makes sense, we should use `add_input_output` and `remap` at the end of the computation to put the registers back into place.

4.4 Classically Controlled Operations

In Figure 11, we give an example of a code using classical bits to control quantum gates. This is the uncomputation of the AND gate, copied from the library. There is a single controlled gate in this code, which is added using `append_basic_gate_controlled`. In general a classically-controlled circuit can be appended using `append_controlled`.

Remark 4. Notice the `validate_input` function, specifying that the circuit only applies to a `BitVector` whose last bit is the AND of the two first bits, and the `define_inverse` function, which defines manually the circuit's inverse (the original `AndGate` circuit). This manual definition will be used when calling `.inverse()`.

```

3  from qarton.binary_operations import AndGate
4  from qarton.circuit import CZ_GATE, BitVector, Circuit, dummify
5
6
7  @dummify
8  class AndGateUncompute(Circuit[BitVector, BitVector]):
9      def __init__(self) -> None:
10         super().__init__()
11         qreg = self.add_input_output(3)
12         self.h(qreg[2])
13         creg = self.get_classical(1)
14         self.msr(qreg[2], creg[0])
15         self.append_basic_gate_controlled(CZ_GATE, qreg[0], qreg[1],
16         ↪ controls=[creg[0]])
17         self.release_classical(creg)
18
19     def define_inverse(self) -> Circuit:
20         return AndGate()
21
22     def validate_input(self, args: BitVector) -> bool:
23         x = args
24         return x[2] == x[0] & x[1]
25
26     def dummy_classical_function(self, args: BitVector) -> BitVector:
27         x = args
28         return BitVector(x[0], x[1], 0)

```

Figure 11: Uncomputation of the AND gate.

In this computation, the CZ gate is applied at runtime only when the measurement output is 1. This corrects the phase and ensures that the AND is properly uncomputed. The `AndGateUncompute` circuit is a simple example of measurement-based uncomputation.

5 Datatypes and Generics

This section describes some additional features of Qarton related to datatypes. The code given in Figure 12 shows how to define a new `DataType`, access its fields as registers, and construct a circuit with a generic type.

```

3  from __future__ import annotations
4  from typing import Generic, TypeVar
5  from qarton.circuit import *
6
7  T = TypeVar("T", bound=QartonObject)
8
9
10 class MyPoint(Generic[T], QartonCustomObject):
11     def __init__(self, x: T, y: T) -> None:
12         self.x, self.y = x, y
13
14     def __hash__(self) -> int:
15         return hash((self.x, self.y))
16
17
18 class MyPointType(DataType[MyPoint[T]], Generic[T]):
19     def __init__(self, d: DataType[T]) -> None:
20         super().__init__()
21         self._d = d
22         self._d_length = (self._d).register_length()
23
24     def to_register(self, o: MyPoint[T]) -> BitVector:
25         return self._d.to_register(o.x) + self._d.to_register(o.y)
26
27     def from_register(self, l: BitVector) -> MyPoint[T]:
28         return MyPoint(
29             self._d.from_register(l[0 : self._d_length]),
30             self._d.from_register(l[self._d_length :]),
31         )
32
33     def register_length(self) -> int:
34         return 2 * self._d_length
35
36     def get_sub_register(self, r: Register, key: RegisterKey) -> Register:
37         if isinstance(key, str):
38             if key == "x":
39                 return qc_reg_cast(r[0 : self._d_length], datatype=self._d)
40             elif key == "y":
41                 return qc_reg_cast(r[self._d_length :], datatype=self._d)
42             raise RegisterIndexError(key)
43
44
45 class SplitPoint(Circuit[MyPoint[T], tuple[T, T]]):
46     def __init__(self, d: DataType[T]) -> None:
47         super().__init__()
48         r = self.add_input(MyPointType(d))
49         self.add_output(r.a["x"])
50         self.add_output(r.a["y"])
51
52     def dummy_classical_function(self, args: MyPoint[T]) -> tuple[T, T]:
53         return args.x, args.y

```

Figure 12: Definition of a generic “point” type. [\[Link to code\]](#)

5.1 Construction of New Objects and Types

Circuit inputs and outputs must be “Qarton objects”. The Python type definition of `QartonObject` is:

```

1  type QartonObject = (
2      UInt | QartonBool | QartonBytes | QartonCustomObject |
3      tuple[QartonObject, ...]
4  )

```

Where `UInt` is an alias to a Python `int`, `QartonBool` is an alias to a Python `int`, and `QartonBytes` is an alias to a Python `str`.

New Qarton objects can be created by subclassing `QartonCustomObject`. In Figure 12, we are creating a new generic class `MyPoint` which can be a pair of any Qarton objects.

Next, we define a `MyPointType` which is a `DataType[MyPoint[T]]`, i.e., a datatype for objects of the class `MyPoint`, which themselves depend on a type variable `T`. To create the datatype, we simply pass as argument another `DataType` instance for objects of type `T`. Then, we need to define the following methods:

- `register_length`, which is the length of the register used to represent objects of this type (here it depends on the element type).
- `to_register` and `from_register`, which convert to and from a `BitVector` object (the actual bit-string in the register). Here, we simply use the element type to convert the elements and concatenate the bit-strings.
- `get_sub_register`, which is a convenient method to access sub-registers of a register having this type. Here, we provide access to sub-registers “x” and “y” for the coordinates of the point. We obtain the sub-registers using Python list syntax and use `qc_reg_cast` to give them the proper datatype.

The circuit `SplitPoint` takes as input a `MyPoint` object and returns a pair of objects. It is also instantiated using a `DataType`. Observe that the `get_sub_register` defined in the type allows us to operate directly on the register `r`: by calling `r.a[index]`, we are actually calling `get_sub_register(r, index)` and accessing the sub-register indexed by `index`. In our case, “x” and “y”. This allows to work more easily with registers having complex types, like tuples, elliptic curve points, and so on.

Circuit subclasses can have variable types and can take datatypes as arguments. Therefore we have defined `SplitPoint` as a generic circuit.

5.2 Completing the Definition of MyPointType

In order to access all the functionalities of Qarton, we can complete our code as shown in Figure 13.

We have added the following:

- In the `MyPointType`, functions which iterate over the values of the object, and which give a default object, or a random value for the object.
- In the `SplitPoint` circuit, a dummy classical function.

Together, this allows us to use the `auto_test` function (provided in `qarton.tests`), which will take a circuit, run it over all valid inputs, and check that the outputs match the dummy classical function:

```

3 from __future__ import annotations
4 import itertools
5 from collections.abc import Iterator
6 from typing import Generic, TypeVar
7 from qarton.circuit import *
8 from qarton.arithmetic import *
9 from qarton.tests import *
10
11 try:
12     from point import MyPoint
13 except ImportError:
14     from .point import MyPoint
15
16 T = TypeVar("T", bound=QartonObject)
17
18
19 class MyPointType(DataType[MyPoint[T]], Generic[T]):
20     def __init__(self, d: DataType[T]) -> None:
21         super().__init__()
22         self._d = d
23         self._d_length = self._d.register_length()
24
25     def to_register(self, o: MyPoint[T]) -> BitVector:
26         return self._d.to_register(o.x) + self._d.to_register(o.y)
27
28     def from_register(self, l: BitVector) -> MyPoint[T]:
29         return MyPoint(
30             self._d.from_register(l[0 : self._d_length]),
31             self._d.from_register(l[self._d_length :]),
32         )
33
34     def register_length(self) -> int:
35         return 2 * self._d_length
36
37     def get_sub_register(self, r: Register, key: RegisterKey) -> Register:
38         if isinstance(key, str):
39             if key == "x":
40                 return qc_reg_cast(r[0 : self._d_length], datatype=self._d)
41             elif key == "y":
42                 return qc_reg_cast(r[self._d_length :], datatype=self._d)
43             raise RegisterIndexError(key)
44
45     def default_value(self) -> MyPoint[T]:
46         return MyPoint(self._d.default_value(), self._d.default_value())
47
48     def random_value(self) -> MyPoint[T]:
49         return MyPoint(self._d.random_value(), self._d.random_value())
50
51     def iterate_values(self) -> Iterator[MyPoint[T]]:
52         for x, y in itertools.product(self._d.iterate_values(), repeat=2):
53             yield MyPoint(x, y)

```

Figure 13: Complete definition of our generic “point” type. [\[Link to code\]](#)

```

1 qc = SplitPoint(UIntType(2))
2 auto_test(qc)

```

`auto_test` tries all valid inputs by default, but we can also pass an argument `nbr` to make a certain number of random tests. Random inputs are selected according to the datatypes' `random_value` functions.

```

1 qc = SplitPoint(UIntType(100))
2 auto_test(qc, nbr=20)

```

Remark 5 (Input / output validation). If the circuit has a `validate_input` function, then `auto_test` will pick inputs at random until they pass the input validation, only then simulate the circuit and repeat until `nbr` valid inputs have been tried. Thus if the validation is very restrictive, the test may become computationally infeasible.

6 Functions and Backends

When programming quantum circuits, we often have several choices for the same functionality. That is, there are several circuits with the same template that implement the same function. We will call such circuits “backends”. When dealing with multiple backends, we will define a *backend handler* (e.g., the “addition of unsigned integers”).

If one only deals with fixed circuits, which always have the same implementation, it suffices to instantiate sub-circuits and rely on `append` as shown in the previous sections. However, making circuits handle backends is very useful to study fine-grained time-memory trade-offs, as we show in this section.

```

3 from qarton.arithmetic import qc_add_uint, CDKMAdder
4 from qarton.circuit import *
5
6 qc = Circuit()
7 x = qc.add_input_output(UIntType(4))
8 y = qc.add_input_output(UIntType(4))
9 qc.append(CDKMAdder(4), x, y)
10 assert qc.simulate((UInt(3), UInt(2))) == (UInt(3), UInt(5))
11
12 qc = Circuit()
13 x = qc.add_input_output(UIntType(4))
14 y = qc.add_input_output(UIntType(4))
15 qc_add_uint(x, y)
16 assert qc.simulate((UInt(3), UInt(2))) == (UInt(3), UInt(5))

```

Figure 14: Using a circuit and a backend handler.

In Figure 14, we define two equivalent circuits (with the same functionality). The first one simply appends an instance of `CDKMAdder`. The second one calls the `qc_add_uint BackendHandler`. This handler takes as input two registers of type `UIntType`, instantiates an in-place adder circuit of appropriate size (determined from the registers), and appends it to the parent circuit. Using `qc_add_uint` simplifies the code, since the instantiation of the adder circuit is now implicit. But the main usefulness of backend handlers is that they allow to *switch* between implementations. In the case of `qc_add_uint`, this allows us to switch between different adders: `GidneyAdder`, `CDKMAdder`, `HybridAdder`, etc. As shown in this section, we can even create a new adder and specify that we want to use it instead.

6.1 Defining Backend Handlers

We explain the definition of backend handlers using the `qc_add_uint` example. In general, when a `qc_*` function should refer to multiple circuit implementations, we need to define a `BackendHandler` object. Otherwise, it can be defined as a standard function.

In our case, the `qc_add_uint` object is:

```
1 qc_add_uint = BackendHandler[[UInt | Register, Register], None]("qc_add_uint")
```

The type parameter `[UInt | Register, Register]` specifies the input of the function, and `None` is the output: we do not return any registers, since the addition operates in-place (and the registers in input are preserved). We can also note that the function's first input can be a `UInt`, because we can use `qc_add_uint` to add a *constant* into a register, with the same syntax (but calling different circuits).

Next, we can register a backend (a circuit class) to the handler, either as a default, or not, by calling: `add_default` or `add` using either a class name, or a path.

```
1 qc_add_uint.add_default("qarton.arithmetic.adder.GidneyAdder")
2 qc_add_uint.add(GidneyAdder)
```

The last remaining step is to define how to connect a backend to its handler. When `qc_add_uint` is called, the `BackendHandler` attempts to instantiate one of its backends. Not all backends may be admissible for a given input: for example, the class `TTKAdder` is only admissible if both inputs are registers of the same size; there is another one to handle inputs of different sizes.

This connection is done by “on-call” functions, which are defined in a class method of the circuit class. We give a complete example in [Figure 15](#), where we define a new backend for addition.

If a circuit needs to be registered as a backend, we shall redefine its class method `initialize_backend_rules`. Inside, we define unnamed functions which we decorate with `define_on_call`. These are the “on-call” functions of the backend handler, which take as input the registers passed, and return either:

- `None` if they cannot handle this input;
- Or a `PreCircuit` object, optionally with an `int` or a `tuple[int, ...]` specifying which registers to return (by default, like in our example, we return no new registers).

Intuitively, the “on-call” function could have returned a `Circuit` instance, but this would lead to an issue: which backends should we use to construct this sub-circuit? The class `PreCircuit` solves this issue by allowing us to *define a circuit instance before it is constructed*. It takes the `Circuit` subclass, the arguments of `__init__` for this subclass, and whether we should invert the circuit or not.

In the code of [Figure 15](#), we check that after calling `qc_add_uint.add`, our new backend is registered. There are other ways to do this. We can call the function using a qualified path to the class instead (`qc_add_uint.add("__main__.TotallyNewAdder")`) or we can also call `TotallyNewAdder.initialize_backend_rules()`, which may be more convenient.

6.2 Specifying Backends

We can now use the function `qc_add_uint` in circuit constructions; and having registered our new backend, we should be able to use it automatically.

```

3  from qarton.arithmetic import (
4      AbstractAdder,
5      TTKAdder,
6      qc_add_uint,
7  )
8  from qarton.circuit import *
9
10
11  class TotallyNewAdder(AbstractAdder):
12      def __init__(self, bit_size: int) -> None:
13          super().__init__(bit_size)
14          x = self.add_input_output(UIntType(bit_size))
15          y = self.add_input_output(UIntType(bit_size))
16          self.append(TTKAdder(bit_size), x, y)
17
18      @classmethod
19      def initialize_backend_rules(cls) -> None:
20
21          @define_on_call(cls, qc_add_uint)
22          def _(x: Register | UInt, y: Register) -> OnCallOutputType:
23              if not isinstance(x, Register):
24                  return None
25              if x.datatype != y.datatype:
26                  return None
27              return PreCircuit(cls, x.datatype.register_length(), inverse=False)
28
29
30  qc_add_uint.add(TotallyNewAdder)
31  # TotallyNewAdder.initialize_backend_rules() # equivalent
32  qc_add_uint.initialize()
33  assert TotallyNewAdder in qc_add_uint.qcs_on_call()

```

Figure 15: Defining a new backend. [\[Link to code\]](#)

Any subclass of `Circuit` accepts a parameter `backends` of type `BackendSpecifier`. This is the case even if the constructor does not explicitly specify it (in our code, we only add it to avoid the complaints of the static type checker). The default value of `backends` is `EmptyBackendSpecifier`.

Qarton supports two types of `BackendSpecifier`:

- **PriorityBackends** stores a set of circuit classes to use in priority. These classes will replace the default backends when applicable. [Figure 16](#) shows an example, where we replace the default adder by our new backend.
- **MemoryConstraint** specifies a *maximal amount of clean ancillas* to use by a circuit. It does not take into account classical bits and dirty ancillas. When setting this as backend specifier, the `BackendHandler` will try to instantiate any available backend, and keep the one which does not exceed the allowed ancilla budget, and minimizes a given constraint (the default is to minimize the count CCX gates, which is the most common usage, but we can also minimize depth). In addition, we can also specify some priority choices inside, which bypass the general rule whenever they can be instantiated, and some excluded sub-circuits or gates.

The choice of backends follows deterministically from the circuit's structure, the available backends, and the `backends` parameter.

```

36 qc = Circuit(backends=PriorityBackends(TotallyNewAdder))
37 x = qc.add_input_output(UIntType(4))
38 y = qc.add_input_output(UIntType(4))
39 qc_add_uint(x, y)
40 qc.freeze()
41
42 # we check that the backend has been used
43 r = ResourceReporter(qc)
44 if __name__ == "__main__":
45     assert "__main__.TotallyNewAdder" in r.sub_circuits_by_class()

```

Figure 16: Using `PriorityBackends` to select our new backend.

Remark 6. As a rule of thumb, `PriorityBackends` is a cheap, low functionality, while `MemoryConstraint` is versatile, but will take more time to get circuits constructed. Indeed, each time multiple backends are available, we must instantiate all of them to compare them and select the best one. In addition, `MemoryConstraint` can lead to failures if no available backend satisfies the constraints. It also has some corner cases which are unsupported (although most circuits, and all circuits from the library, work correctly).

Remark 7. `MemoryConstraint` only modifies the circuit construction by switching between backends. There are many cases where we could instead modify the circuit logic. In the Qarton library, this is done in `HybridAdder` (and similar constructions), which uses more or fewer ancilla qubits depending on the `MemoryConstraint` that was given as input.

`MemoryConstraint` is useful both for its memory-optimization feature, but also because of its secondary rules. [Figure 17](#) is an example where we use it to construct arithmetic circuits *without measurements*, by excluding the MSR gate from constructions. Note that the `exclude` parameter is specified by an object of type `PriorityBackends` (which behaves here like a frozen set of circuit classes), although the naming appears to be a bit unfortunate.

```

3 from qarton.arithmetic import Multiplier
4 from qarton.circuit import MSR, MemoryConstraint, PriorityBackends,
  ↪ ResourceReporter
5
6 qc = Multiplier(5, 5)
7 assert ResourceReporter(qc).measure_depth() == 25
8 qc = Multiplier(5, 5, backends=MemoryConstraint(None,
  ↪ exclude=PriorityBackends(MSR)))
9 assert ResourceReporter(qc).measure_depth() == 0

```

Figure 17: Using MemoryConstraint to exclude measurements.

6.3 Inheriting Backends

We need to adapt the code to work with backends. The main issue is sub-circuits: when creating a circuit with given backends, its sub-circuits will have to inherit adapted backends:

- For PriorityBackends, this is simple: we just pass the same backends to the sub-circuits.
- For MemoryConstraint, this is complex: we need to update the constraint on the number of ancillas that sub-circuits can use, depending on the current memory layout of the circuit.

Furthermore, these backends have to be determined *before* instantiating and appending the sub-circuit. Therefore, we cannot just create this instance ourselves.

Any BackendHandler always passes the appropriate backends to the sub-circuit that it instantiates. If we are not using a BackendHandler, the solution is to define a PreCircuit object instead of a Circuit, and use it in append as we would use a Circuit. An example is given in Figure 18, where we append a Multiplier instance using a PreCircuit. By doing so, we ensure that the low-memory backends we wish for are correctly selected within the Multiplier circuit. We can also use a PreCircuit in append_iterated and append_controlled.

```

3 from qarton.arithmetic import Multiplier
4 from qarton.circuit import *
5
6 if __name__ == "__main__":
7     qc = Circuit(backends=MemoryConstraint(9))
8     x = qc.add_input_output(UIntType(4))
9     y = qc.add_input_output(UIntType(4))
10    pre = PreCircuit(Multiplier, 4, 4, inverse=False)
11    _, _, z = qc.append(pre, x, y)
12    qc.add_output(z)
13    qc.freeze()
14
15    # This is not using the default backends of Multiplier,
16    # which have more ancillas. We end up using the CDKM adder.
17    assert qc.nbr_input_anc() <= 9

```

Figure 18: Using PreCircuit.

Looking back at the Multiplier circuit which we defined in Figure 2, we will have to do the following modifications to handle backends properly, shown in Figure 19:

- We replace the circuit `ControlledTTKCarryAdder` by a call to a `BackendHandler` which contains all our implementations of the same functionality (controlled adder with an outgoing carry bit);
- We replace the explicit `Shift` circuits by our `qc_shift_right` function (although the library only has a single implementation of `Shift` at the moment, so this doesn't change anything);
- We replace the explicit instantiation of `_MultiplierIterate` by a deferred instantiation with `PreCircuit`.

```

3  from qarton.arithmetic import *
4  from qarton.binary_operations import *
5  from qarton.circuit import *
6
7
8  class _MultiplierIterate(InPlaceCircuit[tuple[UInt, UInt, UInt]]):
9      def __init__(self, bit_size_x: int, bit_size_y: int) -> None:
10         super().__init__()
11         x_reg = self.add_input_output(UIntType(bit_size_x))
12         y_reg = self.add_input_output(UIntType(bit_size_y))
13         out_reg = self.add_input_output(UIntType(bit_size_y + bit_size_x))
14
15         qc_caddcarry(
16             qc_bool_reg(x_reg[0]),
17             y_reg,
18             qc_reg_cast(out_reg[0:bit_size_y], datatype=UIntType(bit_size_y)),
19             qc_bool_reg(out_reg[bit_size_y]),
20         )
21         qc_shift_right(x_reg, -1)
22         qc_shift_right(out_reg, -1)
23
24
25  @dummify
26  class Multiplier(Circuit[tuple[UInt, UInt], tuple[UInt, UInt, UInt]]):
27      def __init__(self, bit_size_x: int, bit_size_y: int) -> None:
28         super().__init__()
29         x_reg = self.add_input_output(UIntType(bit_size_x))
30         y_reg = self.add_input_output(UIntType(bit_size_y))
31         out_reg = self.add_anc_output(UIntType(bit_size_y + bit_size_x))
32
33         self.append_iterated(
34             PreCircuit(_MultiplierIterate, bit_size_x, bit_size_y),
35             x_reg,
36             y_reg,
37             out_reg,
38             iterations=bit_size_x,
39         )
40         qc_shift_right(out_reg, bit_size_x)

```

Figure 19: Multiplier circuit with backends. [\[Link to code\]](#)

We also use `PreCircuit` in many situations where we *build* circuits from their sub-circuits. For example, our QAA constructor for quantum amplitude amplification can take as input a `PreCircuit` instead of a `Circuit`, ensuring that the backend specifiers are respected.

6.4 Example: Space-time Trade-off

Figure 20 shows an example in which we instantiate a fused multiply-add circuit for modular integers, from the `Qarton` library, with a memory constraint. The circuit needs a few clean ancillas, and so the construction will fail if the constraint is too small: we catch this `MemoryConstraintUnsatisfiable` error to determine the minimum number of ancilla qubits for this circuit, according to our available backends.

Running this script gives the curve of Figure 21, where each point represents a circuit. We can see that the gate count is somewhat piecewise affine. This is due to our use of some “hybrid” circuits for some arithmetic steps (e.g., addition), which can use all available ancilla qubits. The sharp decreases correspond to switching between two circuits of very different gate counts, where no “hybrid” has been implemented.

Remark 8. Note that `MemoryConstraint` may fail if one explicitly allocates ancillas in-between sub-circuits. Indeed, the constraint may have filled up the circuit’s ancilla budget *before* this allocation, which is bound to increase the total number of ancillas. This then immediately makes the constraint unsatisfiable.

To avoid this, one should use `get_anc`, which does not allocate new ancillas if it does not need to, instead of `add_anc`, and `release_anc` as soon as possible during the circuit.

6.5 Example: Depth Constraint

With a similar script [\[Link to code\]](#) we also instantiated a fused multiply-add, but this time with the constraint:

```
MemoryConstraint(None, constraint=CCX_DEPTH_CONTEXT)
```

This is a `MemoryConstraint` which has actually no memory constraint (we give `None` as the number of qubits), but tries to minimize the CCX depth (the built-in `CCX_DEPTH_CONTEXT` is a `DepthContext` instance giving depth 1 to CCX gates and 0 to all other gates). We compare the resulting full depth to the default instance. The output is given in Figure 22.

The `Qarton` arithmetic library implements logarithmic-depth adders based on the DKRS adder [DKRS04]. While the depth of the default implementation is very clearly in $\mathcal{O}(n^2)$ for an n -bit prime, we can see that the depth-optimized one switches to the DKRS components and obtains an asymptotically lower depth $\mathcal{O}(n \log n)$. An important note is that the DKRS components are not registered by default. Since there are many arithmetic backends, and a `MemoryConstraint` will always try to instantiate them all, we choose to not automatically register them all; so we expose a function `register_dkrs_backends` that will make the DKRS backends automatically available (though they would be registered if we asked for them in `PriorityBackends`).

7 Applications

The `Qarton` library (version 1.0.2) provides around 300 quantum circuit definitions taken or adapted from the literature, including:

- Basic operations on `BitVectors` like permutations and controlled-shifts;
- Boolean linear algebra and Boolean linear system solving (using the algorithm from [BJ22]);
- Arithmetic on unsigned integers, space-optimized [CDKM04], gate-optimized [Gid18], and some hybrid versions;
- Arithmetic on signed integers, which relies on unsigned arithmetic;

```

3  from pathlib import Path
4
5  try:
6      import matplotlib.pyplot as plt
7  except ImportError:
8      # catch this to avoid issues with pytest and such
9      pass
10
11 from tqdm import tqdm
12
13 from qarton.circuit import (
14     MemoryConstraint,
15     MemoryConstraintUnsatisfiable,
16     ResourceReporter,
17 )
18 from qarton.modular_arithmetic import ModularProductAdd
19
20 if __name__ == "__main__":
21     script_dir = Path(__file__).resolve().parent
22     p = 2**70 - 1
23     x: list[int] = []
24     y: list[int] = []
25     s: set[str] = set()
26     for i in tqdm(range(200)):
27         try:
28             qc = ModularProductAdd(p, backends=MemoryConstraint(i))
29             r = ResourceReporter(qc)
30             x.append(i)
31             y.append(r.ccx_count())
32             set_of_classes = {t for t in r.sub_circuits_by_class()}
33             if list(set_of_classes) != list(s):
34                 # uncomment to display the set of circuit classes used
35                 # print(i, set_of_classes)
36                 s = set_of_classes
37         except MemoryConstraintUnsatisfiable:
38             pass
39
40     plt.plot(x, y, marker="o")
41     plt.xlabel("Ancillas")
42     plt.ylabel("CCX gate count")
43     plt.grid(True)
44     plt.savefig(script_dir / "memory_constraint_modular_mul.pdf")
45     plt.show()

```

Figure 20: Putting a memory constraint on modular fused multiply-add. [\[Link to code\]](#)

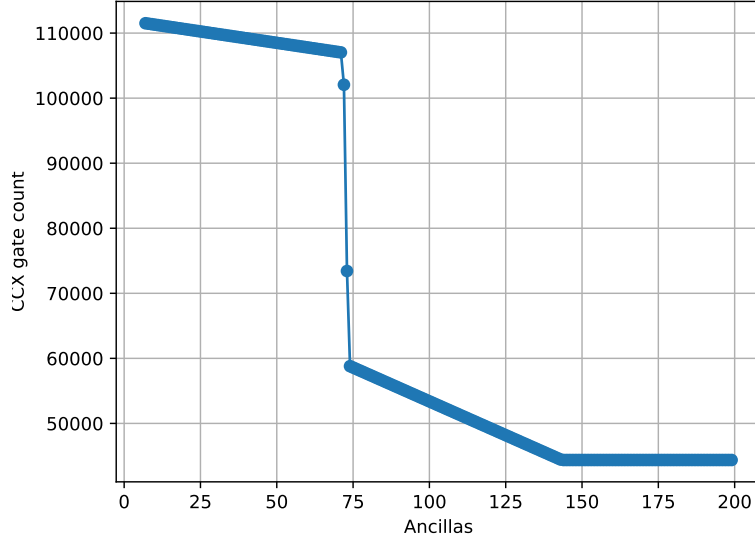


Figure 21: Number of CCX gates depending on the memory constraint for a modular fused multiply-add, for a modulus size of 70 bits.

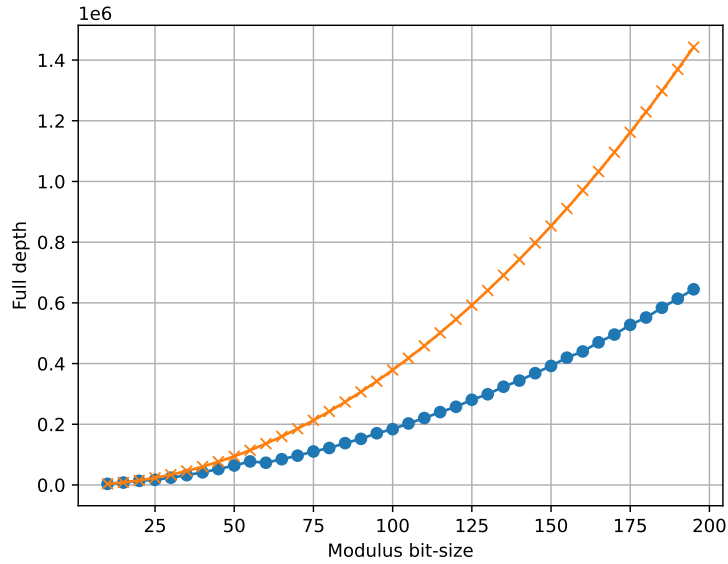


Figure 22: Comparison of full depth between the default construction and the depth-optimized construction of modular multiplication.

- Data structure circuits such as efficient table lookups [BGB⁺18];
- Modular arithmetic circuits, including modular inverse and Fibonacci exponentiation [RV24] (in the standard representation of integers);
- Elliptic curves over prime fields, including point multiplication;
- The AES block cipher, using components from [LL25, LGQW23, SF24];
- Spooky pebbling games on lines [Gid19] and binary trees [KSS25];
- Other circuits such as sorting, Jacobi symbol;
- Generic constructions for Quantum Amplitude Amplification and nested quantum searches using their analysis in [SS24].

Missing from the library are modular arithmetic in Montgomery form, which is used in some implementations of Shor’s algorithm for integer factoring and elliptic curve discrete logarithms [HJN⁺20, GRLR⁺23, Lit23], operations on finite field extensions, and fixed-point and floating-point representations of numbers.

Examples. We also provide the following examples (not part of the library but provided in the repository):

- An implementation of the “textbook” Shor’s algorithm for factoring. The notable feature of this implementation is that it uses the semiclassical Fourier transform [PP00], in which partial measurements of the input register are interleaved with the computations of the Fourier transform, in order to reduce the input register to a single qubit. This also makes the simulation more efficient, as observed in [JH22].
- A re-implementation of the algorithms of [EFS25] for Boolean quadratic system solving. The original code used Qiskit. The `Qarton` re-implementation builds on this by adding types and simplifying memory management. We also use the `QuantumNestedSearch` circuit constructor (a more experimental feature) to explicitly build the circuits for the two nested search algorithms provided in the paper, following the framework of [SS24], and analyze them.
- A re-implementation of the algorithms of [CFS24] for Boolean matrix inversion based on Wiedemann’s algorithm. The reversible Berlekamp-Massey from [CFS24] is included in the `Qarton` library, and the other circuits reproduce the original Qiskit code by adding types and simplifying memory management.

Compatibility. The lack of interoperability between frameworks is also an issue when designing a complex quantum circuit, since components may be already implemented, but in a different framework. To tackle this issue, we have started implementing compatibility features between `Qarton`, Qiskit [JTK⁺24] and OpenQASM3 [CJAA⁺22]. For now, we can export most `Qarton` circuits to valid OpenQASM code, but in the other direction, we only support a very limited subset of OpenQASM (basically allowing to read our own circuits). Simple circuits can also be exported back and forth from Qiskit.

8 Conclusion

A high-level, performant quantum programming language allowing to represent both medium-size and very large quantum circuits, but also to performed detailed resource trade-offs, is a prerequisite for cost estimates in quantum cryptanalysis. `Qarton` is an

attempt at realizing such a tool. Future work will focus on expanding the library of circuits, improving its documentation and support, and providing built-in low-level optimization features.

Acknowledgements. We would like to thank the direct and indirect contributors to the Qarton library of circuits: Clémence Chevnard, Thibault Didier, Quentin Edme, Pierre-Alain Fouque, Aurel Pichollet–Mugnier, as well as Gilles Barthe, Xavier Bonnetain, Jahan Claes, Ulysse Dupont-Latapie, Ludovic Perret, Alessandro Sferlazza and Michael Walter for discussions, bug reports, ideas and feature requests.

This work has been supported by the French Agence Nationale de la Recherche through the QATS project under Contract ANR-24-CE39-7894-01, through the France 2030 program under grant agreement No. ANR-22-PETQ-0008 PQ-TLS, and by the AID through the ASCON-CAT project.

AI Tools. During the writing of the library on versions $\geq 1.0.0$, we did experiment with AI tools for help to refactor and edit the code and documentation, mostly Claude Sonnet 5. Some intermediate versions of the library contained purely AI-generated modules which have since been removed; all the code currently available was authored by humans.

References

- [ASAM18] Mishal Almazrooie, Azman Samsudin, Rosni Abdullah, and Kussay N. Muttar. Quantum reversible circuit of AES-128. *Quantum Inf. Process.*, 17(5):112, 2018. URL: <https://doi.org/10.1007/s11128-018-1864-3>, doi:10.1007/S11128-018-1864-3.
- [BCSS23] Xavier Bonnetain, André Chailloux, André Schrottenloher, and Yixin Shen. Finding many collisions via reusable quantum walks: Application to lattice sieving. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 221–251. Springer, Cham, April 2023. doi:10.1007/978-3-031-30589-4_8.
- [BGB⁺18] Ryan Babbush, Craig Gidney, Dominic W Berry, Nathan Wiebe, Jarrod McClean, Alexandru Paler, Austin Fowler, and Hartmut Neven. Encoding electronic spectra in quantum circuits with linear t complexity. *Physical Review X*, 8(4):041015, 2018.
- [BHMT02] Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305:53–74, 2002.
- [BHT98] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. In Claudio L. Lucchesi and Arnaldo V. Moura, editors, *LATIN 1998*, volume 1380 of *LNCS*, pages 163–169. Springer, Berlin, Heidelberg, April 1998. doi:10.1007/bfb0054319.
- [BJ22] Xavier Bonnetain and Samuel Jaques. Quantum period finding against symmetric primitives in practice. *IACR TCHES*, 2022(1):1–27, 2022. doi:10.46586/tches.v2022.i1.1-27.
- [BLMP19] Daniel J. Bernstein, Tanja Lange, Chloe Martindale, and Lorenz Panny. Quantum circuits for the CSIDH: Optimizing quantum evaluation of isogenies. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 409–441. Springer, Cham, May 2019. doi:10.1007/978-3-030-17656-3_15.

- [BSS22] Xavier Bonnetain, André Schrottenloher, and Ferdinand Sibleyras. Beyond quadratic speedups in quantum attacks on symmetric schemes. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 315–344. Springer, Cham, May / June 2022. doi:10.1007/978-3-031-07082-2_12.
- [BZG⁺26] Ryan Babbush, Adam Zalcman, Craig Gidney, Michael Broughton, Tanuj Khattar, Hartmut Neven, Thiago Bergamaschi, Justin Drake, and Dan Boneh. Securing elliptic curve cryptocurrencies against quantum vulnerabilities: Resource estimates and mitigations. *arXiv preprint arXiv:2603.28846*, 2026.
- [CDKM04] Steven A Cuccaro, Thomas G Draper, Samuel A Kutin, and David Petrie Moulton. A new quantum ripple-carry addition circuit. *arXiv preprint quant-ph/0410184*, 2004.
- [CFS24] Clémence Chevignard, Pierre-Alain Fouque, and André Schrottenloher. Reducing the number of qubits in quantum information set decoding. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part VIII*, volume 15491 of *LNCS*, pages 299–329. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0944-4_10.
- [CFS25] Clémence Chevignard, Pierre-Alain Fouque, and André Schrottenloher. Reducing the number of qubits in quantum factoring. In Yael Tausman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part II*, volume 16001 of *LNCS*, pages 384–415. Springer, Cham, August 2025. doi:10.1007/978-3-032-01878-6_13.
- [CJAA⁺22] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S Bishop, Steven Heide, Colm A Ryan, Prasahnt Sivarajah, John Smolin, Jay M Gambetta, et al. Openqasm 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, 2022.
- [CL21] André Chailloux and Johanna Loyer. Lattice sieving via quantum random walks. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part IV*, volume 13093 of *LNCS*, pages 63–91. Springer, Cham, December 2021. doi:10.1007/978-3-030-92068-5_3.
- [CLM⁺18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An efficient post-quantum commutative group action. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 395–427. Springer, Cham, December 2018. doi:10.1007/978-3-030-03332-3_15.
- [DKRS04] Thomas G Draper, Samuel A Kutin, Eric M Rains, and Krysta M Svore. A logarithmic-depth quantum carry-lookahead adder. *arXiv preprint quant-ph/0406142*, 2004.
- [DLP26] Ulysse Dupont-Latapie and Ludovic Perret. Quapla: a library for exact linear algebra on quantum computers. In *Proceedings of the 6th International Workshop on Programming Languages for Quantum Computing (PLanQC 2026)*, 2026. Poster presentation. URL: <https://popl26.sigplan.org/details/planqc-2026-papers/23/QUAPLA-a-Library-for-Exact-Linear-Algebra-on-Quantum-Computers>.

- [EFS25] Quentin Edme, Pierre-Alain Fouque, and André Schrottenloher. Faster quantum algorithms for MQ2 and applications. *CiC*, 2(1):17, 2025. doi: [10.62056/anjbkscuc2](https://doi.org/10.62056/anjbkscuc2).
- [Gid18] Craig Gidney. Halving the cost of quantum addition. *Quantum*, 2:74, 2018.
- [Gid19] Craig Gidney. Spooky pebble games and irreversible uncomputation. <https://algassert.com/post/1905>, 2019. Algassert blog. Accessed: 2026-03-11.
- [Gid25] Craig Gidney. How to factor 2048 bit RSA integers with less than a million noisy qubits. *arXiv preprint arXiv:2505.15917*, 2025.
- [GLR⁺13] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 333–342, 2013.
- [GLRS16] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying Grover’s algorithm to AES: Quantum resource estimates. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, pages 29–43. Springer, Cham, 2016. doi: [10.1007/978-3-319-29360-8_3](https://doi.org/10.1007/978-3-319-29360-8_3).
- [GRLR⁺23] Élie Gouzien, Diego Ruiz, Francois-Marie Le Régent, Jérémie Guillaud, and Nicolas Sangouard. Performance analysis of a repetition cat code architecture: Computing 256-bit elliptic curve logarithm in 9 hours with 126 133 cat qubits. *Physical review letters*, 131(4):040602, 2023.
- [Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *28th ACM STOC*, pages 212–219. ACM Press, May 1996. doi: [10.1145/237814.237866](https://doi.org/10.1145/237814.237866).
- [HH25] Kengo Hirata and Chris Heunen. Qurts: Automatic quantum uncomputation by affine types with lifetime. *Proceedings of the ACM on Programming Languages*, 9(POPL):155–182, 2025.
- [HJN⁺20] Thomas Häner, Samuel Jaques, Michael Naehrig, Martin Roetteler, and Mathias Soeken. Improved quantum circuits for elliptic curve discrete logarithms. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 425–444. Springer, Cham, 2020. doi: [10.1007/978-3-030-44223-1_23](https://doi.org/10.1007/978-3-030-44223-1_23).
- [HKY⁺24] Matthew P. Harrigan, Tanuj Khattar, Charles Yuan, Anurudh Peduri, Noureldin Yosri, Fionn D. Malone, Ryan Babbush, and Nicholas C. Rubin. Expressing and analyzing quantum algorithms with qualtran, 2024. URL: <https://arxiv.org/abs/2409.04643>, arXiv:2409.04643.
- [HS22] Zhenyu Huang and Siwei Sun. Synthesizing quantum circuits of AES with lower T-depth and less qubits. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part III*, volume 13793 of *LNCS*, pages 614–644. Springer, Cham, December 2022. doi: [10.1007/978-3-031-22969-5_21](https://doi.org/10.1007/978-3-031-22969-5_21).
- [JBK⁺25] Kyungbae Jang, Anubhab Baksi, Hyunji Kim, Gyeongju Song, Hwajeong Seo, and Anupam Chattopadhyay. Quantum analysis of AES. *CiC*, 2(1):25, 2025. doi: [10.62056/ay11zo-3y](https://doi.org/10.62056/ay11zo-3y).

- [JH22] Samuel Jaques and Thomas Häner. Leveraging state sparsity for more efficient quantum simulations. *ACM Trans. Quantum Comput.*, 3(3):15:1–15:17, 2022. doi:[10.1145/3491248](https://doi.org/10.1145/3491248).
- [JNRV20] Samuel Jaques, Michael Naehrig, Martin Roetteler, and Fernando Virdia. Implementing Grover oracles for quantum key search on AES and LowMC. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 280–310. Springer, Cham, May 2020. doi:[10.1007/978-3-030-45724-2_10](https://doi.org/10.1007/978-3-030-45724-2_10).
- [JTK⁺24] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with qiskit. *CoRR*, abs/2405.08810, 2024. URL: <https://doi.org/10.48550/arXiv.2405.08810>, arXiv:2405.08810, doi:[10.48550/ARXIV.2405.08810](https://doi.org/10.48550/ARXIV.2405.08810).
- [Kay18] Alastair Kay. Tutorial on the quantikz package. *arXiv preprint arXiv:1809.03842*, 2018.
- [KMRVK25] Gregory D. Kahanamoku-Meyer, Seyoon Ragavan, Vinod Vaikuntanathan, and Katherine Van Kirk. The jacobi factoring circuit: Quantum factoring with near-linear gates and sublinear space and depth. In Michal Koucký and Nikhil Bansal, editors, *57th ACM STOC*, pages 1496–1507. ACM Press, June 2025. doi:[10.1145/3717823.3718273](https://doi.org/10.1145/3717823.3718273).
- [KSG⁺25] Tanuj Khattar, Noah Shutty, Craig Gidney, Adam Zalcman, Nouredin Yosri, Dmitri Maslov, Ryan Babbush, and Stephen P Jordan. Verifiable quantum advantage via optimized dqj circuits. *arXiv preprint arXiv:2510.10967*, 2025.
- [KSS25] Niels Kornerup, Jonathan Sadun, and David Soloveichik. Tight bounds on the spooky pebble game: Recycling qubits with measurements. *Quantum*, 9:1636, 2025.
- [KT17] Ghazal Kachigar and Jean-Pierre Tillich. Quantum information set decoding algorithms. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, pages 69–89. Springer, Cham, 2017. doi:[10.1007/978-3-319-59879-6_5](https://doi.org/10.1007/978-3-319-59879-6_5).
- [Kup13] Greg Kuperberg. Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem. In *8th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2013)*, pages 20–34. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013.
- [LGQW23] Zhenqiang Li, Fei Gao, Sujuan Qin, and Qiaoyan Wen. New record in the number of qubits for a quantum implementation of aes. *Frontiers in Physics*, 11:1171753, 2023.
- [Lit23] Daniel Litinski. How to compute a 256-bit elliptic curve private key with only 50 million toffoli gates. *arXiv preprint arXiv:2306.08585*, 2023.
- [LL25] Haoyu Liao and Qingbin Luo. Quantum circuit synthesis for AES with low DW-cost. In Goichiro Hanaoka and Bo-Yin Yang, editors, *ASIACRYPT 2025, Part II*, volume 16246 of *LNCS*, pages 475–507. Springer, Singapore, December 2025. doi:[10.1007/978-981-95-5096-8_15](https://doi.org/10.1007/978-981-95-5096-8_15).

- [LPS20] Brandon Langenberg, Hai Pham, and Rainer Steinwandt. Reducing the cost of implementing the advanced encryption standard as a quantum circuit. *IEEE Transactions on Quantum Engineering*, 1:1–12, 2020.
- [LPZW23] Qun Liu, Bart Preneel, Zheng Zhao, and Meiqin Wang. Improved quantum circuits for AES: Reducing the depth and the number of qubits. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part III*, volume 14440 of *LNCS*, pages 67–98. Springer, Singapore, December 2023. doi:10.1007/978-981-99-8727-6_3.
- [LXX⁺23] Da Lin, Zejun Xiang, Runqing Xu, Shasha Zhang, and Xiangyong Zeng. Optimized quantum implementation of AES. *Quantum Inf. Process.*, 22(9):352, 2023. URL: <https://doi.org/10.1007/s11128-023-04043-9>, doi:10.1007/S11128-023-04043-9.
- [NC10] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.
- [NIS16] NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. URL: <https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf>.
- [PBSV21] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. Unqomp: synthesizing uncomputation in quantum circuits. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 222–236, 2021.
- [Pei20] Chris Peikert. He gives C-sieves on the CSIDH. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 463–492. Springer, Cham, May 2020. doi:10.1007/978-3-030-45724-2_16.
- [PP00] S Parker and Martin B Plenio. Efficient factorization with a single pure qubit and log n mixed qubits. *Physical Review Letters*, 85(14):3049, 2000.
- [RV24] Seyoon Ragavan and Vinod Vaikuntanathan. Space-efficient and noise-robust quantum factoring. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VI*, volume 14925 of *LNCS*, pages 107–140. Springer, Cham, August 2024. doi:10.1007/978-3-031-68391-6_4.
- [SBZ⁺24] Raphael Seidel, Sebastian Bock, René Zander, Matic Petrič, Niklas Steinmann, Nikolay Tcholtchev, and Manfred Hauswirth. Qrisp: A framework for compilable high-level programming of gate-based quantum computers, 2024. URL: <https://arxiv.org/abs/2406.14792>, arXiv:2406.14792.
- [SF24] Haotian Shi and Xiutao Feng. Quantum circuits of AES with a low-depth linear layer and a new structure. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part VIII*, volume 15491 of *LNCS*, pages 358–395. Springer, Singapore, December 2024. doi:10.1007/978-981-96-0944-4_12.
- [SGT⁺18] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q# enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the real world domain specific languages workshop 2018*, pages 1–10, 2018.

- [Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994. doi:[10.1109/SFCS.1994.365700](https://doi.org/10.1109/SFCS.1994.365700).
- [SHT18] Damian S Steiger, Thomas Häner, and Matthias Troyer. Projectq: an open source software framework for quantum computing. *Quantum*, 2:49, 2018.
- [SPMH03] Vivek V Shende, Aditya K Prasad, Igor L Markov, and John P Hayes. Synthesis of reversible logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):710–722, 2003.
- [SS24] André Schrottenloher and Marc Stevens. Quantum procedures for nested search problems: with applications in cryptanalysis. *CiC*, 1(3):9, 2024. doi:[10.62056/ae0fhbm0](https://doi.org/10.62056/ae0fhbm0).
- [TTK10] Yasuhiro Takahashi, Seiichiro Tani, and Noboru Kunihiro. Quantum addition circuits and unbounded fan-out. *Quantum Information and Computation*, 10(9-10):872–890, 2010.
- [VGDV24] Hristo Venev, Timon Gehr, Dimitar Dimitrov, and Martin Vechev. Modular synthesis of efficient quantum uncomputation. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):2097–2124, 2024.
- [ZWS⁺20] Jian Zou, Zihao Wei, Siwei Sun, Ximeng Liu, and Wenling Wu. Quantum circuit implementations of AES with fewer qubits. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 697–726. Springer, Cham, December 2020. doi:[10.1007/978-3-030-64834-3_24](https://doi.org/10.1007/978-3-030-64834-3_24).