Towards Practical Hybrid Quantum / Classical Computing

by

Marcus Edwards

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirements for the degree of
Master of Science
in
Physics (Quantum Information)

Waterloo, Ontario, Canada, 2020

AUTHOR'S DECLARATION

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

STATEMENT OF CONTRIBUTIONS

This thesis is based on the following publications and presentations.

- Chapter 2 is based on:

  Marcus Edwards and Shohini Ghose. "Controlled Teleportation on the IBM Quantum Computing Platform". In: Wilfrid Laurier University 2018 Student Thesis Poster Conference (Mar. 2018). doi: 10.13140/RG.2.2.27396.8896844.

- Chapter 3 is based on:

  Marcus Edwards, Atefeh Mashatan, and Shohini Ghose. "A Review of Quantum and Hybrid Quantum / Classical Blockchain Protocols". In: Quantum Information Processing 19.6 (2020). doi: 10.1007/s11128-020-02672-y.

- Chapter 4 is based on:

  Marcus Edwards, Atefeh Mashatan, and Shohini Ghose. "Quantum Annealers as Continuous Testing Automation Backends for Classical Web Code". In: Ryerson Cybersecurity Research Lab 2019 Research Exhibit (Dec. 2019). doi: 13210.13140/RG.2.2.29258.1120665.

ABSTRACT

Quantum computing is in a critical phase where theoretical schemes and protocols are now being implemented in the real world for the first time. Experimental implementations can help us solidfy ideas, and can also complicate them. In the case of quantum communication protocols, we present the first experimental implementations of two entanglement-based schemes using IBM's superconducting transmon qubit based technology. We find that the schemes are experimentally feasible with current technology, and give an idea of how much room for improvement there is before quantum technology can meet the highest theoretical expectations. These communication schemes may be fundamental components of the future quantum internet. We also present an overview of the emerging field of quantum blockchain protocols that could form a part of the quantum / classical communication structures of the future. Interaction between classical and quantum technologies can impair purely quantum designs, but can also be harnessed to enhance hybrid quantum / classical approaches. Finally, we suggest a path towards the hybridization of arbitrary code execution and verification in the hybrid quantum / classical networks of the future.

ACKNOWLEDGMENT

I would like to thank my supervisors Dr. Shohini Ghose and Dr. Robert Mann for the invitation to participate in higher education, and for the incredible opportunity to work on these projects with them. I am also grateful to Dr. Atefeh Mashatan and Dr. Roger Melko for their helpful guidance and advice. I would like to recognize Josh Izaac for helping me join Xanadu's open source community as I jumped into the world of open source quantum computing software.

DEDICATION

To my wife, who is my center, a remarkably imperturbable source of stability, who has sustained us through incredibly chaotic times.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter One

Introduction

Motivation and Background.

1.1 Motivation for Hybrid Computing

Hybrid quantum / classical computing (or "Quassical" computing [CCD15]) is by its nature a field that focusses on practical applications. Rapid progress in the Noisy Intermediate-Scale Quantum (NISQ) era of quantum computing has led to a situation where researchers and programmers have access to quantum computers that are not yet capable of demonstrating useful quantum speedups in most applications on their own, but in combination with useful classical algorithms are close to real advantage.

One of the key observations that leads to an interest in quantum / classical hybrid algorithms (QCH) is that most quantum algorithms require classical counterparts, at a fundamental level. This means that some non-trivial classical processing is necessary to make use of the quantum algorithm. For example, this may be a pre-processing step which prepares data for a quantum algorithm or a post-processing step which handles data coming from a quantum algorithm. Classical processing of this kind is often required because a quantum algorithm requires a unique quantum state to be prepared. Preparing such a state can be more work than preparing the same input data for a computationally equivalent classical algorithm would require. An example of this is found in Grover's search [Gro96]. Grover's search allows for a search of n entries in an unsorted database in $O(\sqrt{n})$ time, which is less than the linear time required by classical computers which have to check each record individually. This is possible because a quantum computer can operate on superpositions of data. Once a database is described in its entirety by a quantum superposition, then the desired record's amplitude can be tagged and amplified by a quantum process called inversion about the mean.

To build a quantum description of a classical database requires that we both have the classical database on hand and that we transform each classical record to a "quantum record" that is represented as an amplitude in a quantum superposition as a pre-processing step. Creating this superposition is not trivial. This also requires $O(n)$ storage space, and it has

been argued that properly considering the pre-processing step leads to an understanding that a linear time classical search is actually more efficient than what Grover proposed [LU05]. In this case, not properly considering the hybrid nature of the Grover's search algorithm led to an inefficient design.

There are several approaches to QCH algorithm design, and all of them involve quantum and classical processing steps that interact through some interface. This interface can take the form of a classical parameter space for a quantum operator, a quantum-assisted training procedure for a classical neural network, a truly random quantum seed for a classical hash function, or a quantum search of a classical database, to give a few examples.

Various promising QCH algorithms have been proposed and demonstrated, including applications in correlated electron systems simulations [Yao+20], open quantum systems simulations [LSH19], machine learning [BRP18; Vin+19], image processing [SPS19], optimization [Neu+17], graph partitioning [UNM17] and more. A hybridized Grover's search has also been proposed that is an improvement on the classical search, even when considered in the hybrid regime [LU05]. In each case, these algorithms introduce the relatively new advantages of quantum computing to old computing problems.

1.2 Computing with Qubits

The qubit model is the most commonly used in quantum computing. A qubit is a two-level quantum system, meaning that it is described by its association with two unique quantum basis states [Mer07]. The state of a qubit in terms of the standard computational basis states $|0\rangle, |1\rangle$ is generally given by a 2-dimensional vector $|\psi\rangle$, a complex-weighted sum of the basis states.

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle = \begin{bmatrix} \alpha_0 \\ \\ \alpha_1 \end{bmatrix} \tag{1.1}$$

The complex amplitudes α_0 and α_1 are normalized so that any valid pure quantum state lies graphically on a sphere, the "Bloch sphere".

$$|\alpha_0|^2 + |\alpha_1|^2 = 1 \tag{1.2}$$

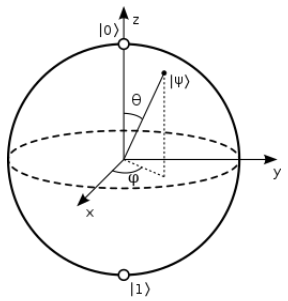

Figure 1.1 The Bloch Sphere

This generalizes in theory to pure states of arbitrarily many qubits Ψ, the state space of which can be visualized as a set of hyperspheres.

$$|\Psi >= \sum_{i=0}^{N} \alpha_i |i > \tag{1.3}$$

$$\sum_{i=1}^{N} |\alpha_i|^2 = 1 \tag{1.4}$$

Quantum computing involves intentionally evolving such quantum states using predictable evolution operators. Since a quantum state is given by a vector, an operator may be represented by a matrix. An operator is applied to a state simply by matrix multiplication.

Valid operators are unitary matrices. A unitary matrix U satisfies the property that $U^\dagger U = UU^\dagger = I$. Here $\dagger$ denotes the conjugate transpose of the matrix U and I is the identity. Unitary matrices are surjective isometries, meaning that they do not change the distance between elements of the state space and preserve its topology (i.e. the topology of the Bloch sphere).

These unitary operators used for quantum computing are not capable of describing all possible evolutions of quantum systems. Rather, they describe the evolutions that we find useful for universal quantum computing. Decoherence from interaction with the environment and physical implementation imperfections can cause non-unitary evolution, which generally makes computation difficult [Zeh70].

Operators are often constructed from a set of basic operators, the Pauli gates $\sigma_x, \sigma_y, \sigma_z$, which form the set of reflections across each axis x, y, z of the Bloch sphere.

$$\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{1.5}$$

An arbitrary single-qubit operator U can be constructed from a weighted sum of the Pauli gates. Such an operator is characterized by a "Bloch vector" v.

$$U = x\sigma_x + y\sigma_y + z\sigma_z \tag{1.6}$$

$$v = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{1.7}$$

The overall states $|\Phi\rangle$ of some groups of subsystems $|\phi_0\rangle, |\phi_1\rangle, \ldots, |\phi_n\rangle$ can be described

succinctly as the tensor product of the individual subsystems.

$$|\Phi\rangle = |\phi_0\rangle \otimes |\phi_1\rangle \otimes \cdots \otimes |\phi_n\rangle \tag{1.8}$$

However, not all states have a simple tensor product structure. Local configurations can yield more complex states. For example, we discuss what is required to construct quantum states involving entanglement at the end of this chapter. For another example, the energy states of D-Wave's quantum annealers are described by a more complex Hamiltonian with orthogonal parts that do not commute [H18].

1.3 Quantum Annealing

D-Wave's approach to quantum computing is an instance of adiabatic quantum computing [McG]. Adiabatic quantum computing is polynomially equivalent to the gate model approach to quantum computing, but the method of encoding problems into the system Hamiltonian in each approach is very different. In both cases there is an initial Hamiltonian H_I, a final Hamiltonian H_F, and a change of state characterized by a Hermitian matrix H_t. In the case of the gate model, H_t describes a number of discrete operators that change the state of the system from its initial state to its final state. These operators together form a quantum circuit that executes a quantum algorithm. In the case of adiabatic quantum computing, on the other hand, H_t describes a gradual adiabatic evolution path from H_I to H_F.

$$H_t = s(t)H_I + (1 - s(t))H_F \tag{1.9}$$

Here $s(t)$ decreases linearly in time from 1 to 0, which causes H_t to evolve from H_I to H_F.

H_I and H_F may describe systems of multiple qubits, as we will see in the case of D-Wave's Hamiltonian. The goal of a programmer is to design a Hamiltonian which will settle to a valid solution to a particular problem as $s(t) \to 0$.

In the model used by D-Wave [Cho08], a strong transverse field $\hat{\sigma}_x$ is applied to all qubits at the beginning of an experiment. This causes the first term in the Hamiltonian to dominate. Fields $\hat{\sigma}_z$ are then applied locally to the individually indexed qubits according to the qubit biases h_i. Here i indexes the qubits. These may also be applied to pairs of qubits, establishing a coupling between them according to coupling strengths $J_{i,j}$. Here i, j index the pairs of interacting qubits.

$$H_{ising} = \frac{A(s)}{2}(\sum_i \hat{\sigma}_x^{(i)}) + \frac{B(s)}{2}(\sum_i h_i \hat{\sigma}_z^{(i)} + \sum_{i>j} J_{i,j} \hat{\sigma}_z^{(i)} \hat{\sigma}_z^{(j)}) \tag{1.10}$$

The transverse field is weakened gradually according to the energy scaling functions $A(s)$ and $B(s)$ so that by the end of the experiment the second term dominates and will finally settle to a classically observable state. Programming such a quantum annealer is equivalent to encoding a problem into the second term such that the system settles into a final state that solves the embedded problem with high probability. This model allows for a quantum speedup over classical annealing methods since in order to settle to stable states a quantum system may use quantum tunneling to take a "shortcut" to a solution.

1.4 Quantum Computing in the Gate Model

Most universal quantum computers follow the gate model of quantum computing. Quantum circuits in this model are easier to visualize due to their being discrete. Operators can be represented pictorially in circuit diagrams. The operators are depicted as being applied to quantum channels, visually horizontal lines, which each depict the evolution of a qubit q_i as

time flows from left to right.

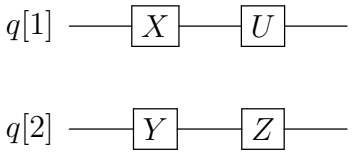

Figure 1.2 Example circuit diagram

A notable operator is the Hadamard gate since it maps the basis states $|0\rangle, |1\rangle$ each to a perfect superposition of basis states $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$, $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$.

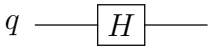

Figure 1.3 Pictorial Hadamard

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{1.11}$$

Another notable operator is the Toffoli (or CCNOT) gate, since it facilitates interaction between qubits.

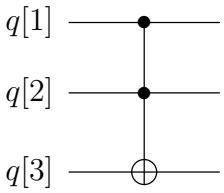

Figure 1.4 Pictorial Toffoli (CCNOT)

$$CCNOT = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.12}$$

The $\frac{\pi}{4}$ rotation operator called the T-gate is also interesting, since the Toffoli, Hadamard and T-gate together form a gateset that is complete for universal computing [Kit97].

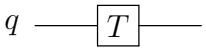

Figure 1.5 Pictorial T-gate

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix} \tag{1.13}$$

There are many universal gatesets, but each generally involves operators that facilitate each of the key ingredients for quantum computing: superposition, interaction and rotation. One such gateset is the set of all single-qubit gates, generally captured by the three-parameter operator U_3, and the two-qubit $CNOT$ gate [Bar+95]. Here, U_3 includes *all* single-qubit rotations, as well as maps involving superimposed states. $CNOT$ facilitates interaction between a pair of qubits. This is more than enough for universal computation.

$$U_3(\theta, \phi, \lambda) = \begin{bmatrix} cos(\frac{\theta}{2}) & -e^{\lambda i} sin(\frac{\theta}{2}) \\ e^{\phi i} sin(\frac{\theta}{2}) & e^{\lambda i + \phi i} cos(\frac{\theta}{2}) \end{bmatrix} \tag{1.14}$$

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.15}$$

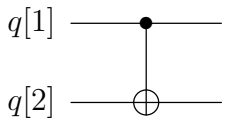

Figure 1.6 Pictorial CNOT

Since a qubit is defined by complex association with the classical Boolean basis states $|0\rangle$ and $|1\rangle$, it may be considered a higher dimensional unit of information than the Boolean bit. The space of functions that are computable using a number of qubits is larger than than the space of computable functions using the same number of bits, since qubits are simply *more general*.

The last element of any quantum computing algorithm is measurement. Computational measurement causes an evolution of a quantum state that results in its being rotated to one of the computational basis states. Generally, the likelihood of a measurement to cause the state to resolve to a particular basis $|i\rangle$ is the same as the square of the corresponding complex coefficient's magnitude $|\alpha_i|^2$.

A quantum effect that is central to the protocols studied in this thesis is quantum entanglement. In any communication protocol, the communication channels and the units

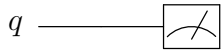

Figure 1.7 Pictorial measurement

of information shared may be considered resources. In quantum communication protocols, entanglement may also be used as a resource. Entanglement can be used to enable multi-party communication and high-density information encodings as in the controlled teleportation and dense coding protocols [LD08; HLG01].

Entanglement may be prepared by the application of $CX_{12}H_1$ to two basis-state qubits. This causes the two qubits to have correlated measurement outcome probability amplitudes.

$$CX_{12}H_1 \left|00\right\rangle = \frac{\left|00\right\rangle + \left|11\right\rangle}{\sqrt{2}} = \left|\phi^+\right\rangle \tag{1.16}$$

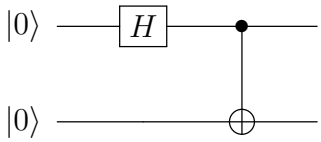

Figure 1.8 Example Entanglement Protocol

Applying the entanglement protocol to each of the four unique initial basis state combinations yields what are known as the four Bell states.

$$|\psi^{\pm}> = \frac{|01> \pm |10>}{\sqrt{2}} \tag{1.17}$$

$$|\phi^{\pm}> = \frac{|00> \pm |11>}{\sqrt{2}} \tag{1.18}$$

Entanglement is useful for quantum communication since the entangled particles retain

their correlation regardless of physical separation, until they are measured. This is the basis of all entanglement-based quantum communication algorithms, which are expected to play a foundational role in the future quantum internet. If the quantum internet is ever going to support useful communication networks and data structures like grid, cloud or blockchain networks then it is likely that the quantum technology employed will need to support entanglement-based communication.

Chapter Two

Quantum Control for Networked Communication

An Experimental Study of the Implementation of Controlled Algorithms using Modern Quantum Computers.

2.1 Chapter Overview

We experimentally implement two quantum communication protocols with controlling parties for the first time in the superconducting quantum computing setting, using the IBMQX4 quantum computer. Controlled teleportation and controlled dense coding are two controlled communication schemes that will be fundamental elements of quantum networking and the quantum internet. The ability for a controller to affect the quantum communication processes in these protocols relies on the details of the underlying quantum computing technology and infrastructure. We demonstrate that using current quantum computing technology, the effectiveness of control is quantifiable despite challenges including decoherence and noise.

2.2 Introduction

The practical usability of any communication technology relies on its operator's ability to control its behaviour. For a digital electrical communication system to work, it is necessary that digital signals can be routed between corrected transmitting and receiving devices, for example. Data compression can also be an important part of signal transmission. In quantum information science efforts, control is of particular importance. While quantum communication promises to vastly improve the efficiency and security of communication through protocols like BB84 quantum key distribution [BB20], these benefits do not come without proportional challenges.

IBM has delivered a superconducting quantum computing system that uses Transmon qubits to achieve an early approximation of general quantum computing with five qubits. While this isn't enough for a user to achieve powerful quantum computing algorithms, it is enough to implement many fundamental quantum information experiments. We are particularly interested in experimentally testing quantum control schemes that relate to

quantum communication. Two quantum algorithms that will enable the communication of quantum states between transmitter and receiver with sufficient fidelity and data density are the Controlled Teleportation [KB] and Controlled Dense Coding [HLG01] schemes. We experimentally implement both of these protocols in this study. Controlled teleportation has recently been demonstrated in the linear optical setting [BČL19], but we believe that this is the first formal implementation of these control schemes in the superconducting setting.

Teleportation promises to enable the transport of state information between two locations with excellent fidelity, while dense coding ensures data density in a transmission. The ability to control the behaviour of a system implementing either protocol will be essential if a practical implementation of a quantum communication system is to be achieved. The first proposal of a controlled teleportation scheme was made by Karlsson and Bourennane in 1998 [KB]. Controlled teleportation can be used in the same way as a quantum state sharing scheme to share a quantum secret among several receivers [GR23].

In 2001, a controlled dense coding scheme was also proposed [HLG01] by Hao, Li and Guo. These two controlled algorithms show how the destination and content of a quantum communication might be influenced by controlling logic. The ability to effectively implement these will be a prerequisite for further practical implementations of quantum communication system elements including quantum channel switches, routers and networks. A quantum network built on these elements would be "ultimately secure", as was shown in 1997 by Biham and Mor [BM97]. This means that secret communications over a correctly implemented quantum network will be impossible for malicious parties to take advantage of; a huge step beyond the security of the modern web.

This chapter explores the effectiveness of controlling parties in controlled teleportation and dense coding algorithms run on the most advanced and cutting-edge quantum computing systems available today. Each algorithm is tested on the IBMQX4 quantum computer and the results compared to theoretical ideals.

We analyze each protocol's resistance to unauthorized message transport and fidelity in

conveying allowed messages. We compare computed results to simulated results obtained using IBM's quantum computing simulation platform. IBM's quantum computing platform is an ideal tool for this experiment since it provides simulation and quantum computation services, the later of which is difficult to find elsewhere. IBM's is the first commercialized cloud quantum computing service [Edi17].

2.3 Teleportation

Quantum teleportation takes advantage of entanglement to achieve the secure sharing of unknown quantum state information between separate sending and receiving locations. One method of performing quantum teleportation is to prepare an entangled state of two qubits and then split the participating qubits up, one to each of the locations. Then the sender, say Alice, performs a Bell measurement involving a qubit with the state she wishes to teleport, and the entangled qubit at her location. Alice performs basis measurements of these same two qubits, and classically transmits the results of these measurements to Bob. Bob may then use this information and his entangled qubit to reconstruct the "teleported" state.

Quantum teleportation has been tested and verified to perform within statistically significant bounds on IBM's quantum computer [Fed08]. An extension of quantum teleportation is controlled teleportation, which introduces a third party, say Charlie. Charlie's role is to allow or disallow Alice and Bob to perform a teleportation successfully. The successful implementation of controlled teleportation allows the introduction of logical control over teleportation operations such that a transmitted state cannot be derived by the receiving location when the operation is not allowed. When a teleportation is allowed, the procedure should perform the same as a simple teleportation procedure without a controller.

This chapter will focus on the validity and extent of Charlie's control over the success of a teleportation operation in a known controlled teleportation procedure [LD08].

2.4 The Controlled Teleportation Procedure

The controlled teleportation procedure used in this study uses one additional qubit on top of the three used in the simple teleportation circuit. Instead of beginning the procedure with a pair of entangled qubits, the procedure begins with the preparation of an entangled state of three qubits, called a Greenberger-Horne-Zelinger state. By performing this preparation, the following GHZ state is created.

$$|\psi_{GHZ}>_{ABC}= \frac{|000>_{ABC} +|111>_{ABC}}{\sqrt{2}} \tag{2.1}$$

Similar to the simple teleportation procedure, the next component of controlled teleportation is for Alice to perform a CNOT from a qubit with the state to be teleported onto the entangled qubit at her location. The unknown state to be teleported will be the state of a fourth qubit.

$$|x>_x= \alpha|0>_x +\beta|1>_x \tag{2.2}$$

Next, Alice applies a Hadamard gate to the qubit with the arbitrary state to be transmitted, x. This achieves a Bell measurement of A and x. A Bell measurement of two qubits is an entangling operation that puts the pair of qubits into one of four states known as the Bell states.

$$|\psi^{\pm}>= \frac{1}{\sqrt{2}}(|0>_x |1>_A \pm|1>_x |0>_A) \tag{2.3}$$

$$|\phi^{\pm}>= \frac{1}{\sqrt{2}}(|0>_x |0>_A \pm|1>_x |1>_A) \tag{2.4}$$

The overall state of the four qubits then takes the following general form.

$$
\begin{aligned}
&|x>_x |\psi_{GHZ}>_{ABC}= \\
&\frac{1}{2}[|\phi^+>_{xA} \otimes(\alpha|00>+\beta|11>)_{BC} \\
&+|\phi^->_{xA} \otimes(\alpha|00>-\beta|11>)_{BC} \\
&+|\psi^+>_{xA} \otimes(\alpha|11>+\beta|00>)_{BC} \\
&+|\psi^->_{xA} \otimes(\alpha|11>-\beta|00>)_{BC}]
\end{aligned} \tag{2.5}
$$

When Alice performs the Bell measurement, x and A will assume one of the four Bell states and qubits B and C will collapse from this general form to a corresponding state with two outcomes of equal likelihood. For example, in the case of the Bell State $|\phi^+>_{xA}$, qubits B and C would collapse into the following state.

$$
\begin{aligned}
&|\psi>_{BC}= (\alpha|00>+\beta|11>)_{BC} = \\
&\frac{1}{\sqrt{2}}[(\alpha|0>+\beta|11>)_B|+x>_C \\
&+(\alpha|0>-\beta|11>)_B|-x>_C]
\end{aligned} \tag{2.6}
$$

In this collapsed state, $|\pm x>$ are the eigenvectors of the x basis. The $|\pm x>$ components make it possible for Charlie to determine which of the two outcomes of equal likelihood has taken place by performing a measurement in the x basis basis of the C qubit. Alice can also measure x and A in order to determine which Bell state has been assumed. If both Charlie and Alice classically transmit the results of these measurements to Bob, it is then possible for Bob to determine what gates need to be applied to qubit B in order for its state to be equal to the transmitted state. Without being informed of the result from Charlie's measurement, Bob cannot know which gate operations to apply; therein lies

Charlie's control of the teleportation operation. Without knowing which operation to apply, Bob must guess, which means his results will not always match what Alice intended. In practice, this introduces a loss of fidelity that can be seen both in computer simulations and in experimental demonstrations of the protocol on the IBMQX4.

The possible operations that Bob may be required to perform are shown in relation to the corresponding measurement results below.

Bell State	Charlie's Result	Bob's Operation
$\|\phi^+>_{xA}$	$\|+x>$	I
$\|\phi^+>_{xA}$	$\|-x>$	Z
$\|\phi^->_{xA}$	$\|+x>$	Z
$\|\phi^->_{xA}$	$\|-x>$	I
$\|\psi^+>_{xA}$	$\|+x>$	X
$\|\psi^+>_{xA}$	$\|-x>$	XZ
$\|\psi^->_{xA}$	$\|+x>$	XZ
$\|\psi^->_{xA}$	$\|-x>$	X

Table 2.1 Decoding operations

The implementation of this protocol can be represented by a quantum circuit with the gate labelled B representing the relevant of the four possible operations for Bob to apply, $q[0]$ as qubit B, $q[1]$ as C, $q[2]$ as A and $q[3]$ as x.

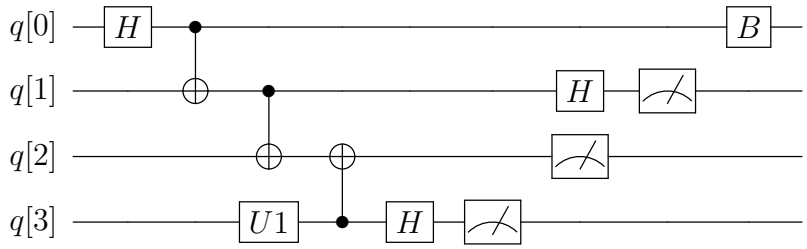

Figure 2.1 Theoretical controlled teleportation circuit

2.5 Implementation on the IBMQX4

In order to demonstrate the true efficacy of Charlie's control in the protocol, the protocol was implemented and tests of effectiveness automated to run on IBM's quantum computing platform. To achieve a true test of the protocol, several limitations of the platform had to be considered. The first obstacle to implementing the protocol was the fact that quantum circuits to be run on the IBM platform must be completely predefined in IBM's own definition language, Open QASM. This means that measurements taken during a circuit execution cannot affect the gates that are applied as a part of the circuit. This clearly conflicts with Bob's need to use the measurement results from Alice and Charlie to inform his final gate operations.

Secondly, the platform API returns a probability distribution of measurement results over a number of executions. This introduces complexity because in the controlled teleportation protocol, each run will involve a different Bell state and collapsed state that define the context for Bob's final operation. Since the result of a run on the IBMQX4 or IBM simulator provides no insight into these contexts and cannot discriminate between them, the overall probability of measuring qubit B as either 1 or 0 over a number of executions is meaningless.

The approach that was taken to get around these limitations was to predefine each of the four possible circuits. Then each time a circuit was executed, the operation done by Bob would be known. A post-selection algorithm was implemented that determined from each set of measurement results whether the measurements made by Charlie and Alice matched

with the operations applied by Bob.

For each of the four Bell states and both of Charlie's possible measurement results, each of the four circuits was run in a batch of 1000 executions each. After these 8000 executions, the post-selection algorithm reported on the relevant probabilities of Bob's outcomes. Then each of the four circuits was run 1000 more times, and the post-selection algorithm was not given Charlie's expectation, instead only filtering out results where Alice's measurement did not correspond with Bob's decoding operations. In order to fully test the functionality of the process, five input states were prepared and the procedure involving a total of 12000 executions was repeated for each one.

As a calibration step preceding these experiments, the full set of executions was performed with a basis state input. These execution results yielded an average fidelity of 90%. We can consider this fidelity to be a baseline that reflects the performance of the circuit itself since these executions involved trivial state preparation and fidelity measurement operations that should not introduce the same gate-level noise as the state preparation and fidelity measurement gates used in the experiments.

In tabular and graphic presentations of data, the following enumeration of the input states will be used.

α	β	Input State
0.71	0.71	$\|\psi_1>$
0.5	0.87	$\|\psi_2>$
0.3	0.95	$\|\psi_3>$
0.37	0.93	$\|\psi_4>$
0.17	0.98	$\|\psi_5>$

Table 2.2 Teleportation input states

To provide a metric for the success of the procedure, the dagger of the input state was applied to qubit B at the end of the procedure, and the qubit then measured. This meant that if working properly, the entire process should have guaranteed that Bob's qubit was always measured to be a 0. This metric is used as the measure of the controlled teleportation protocol's fidelity.

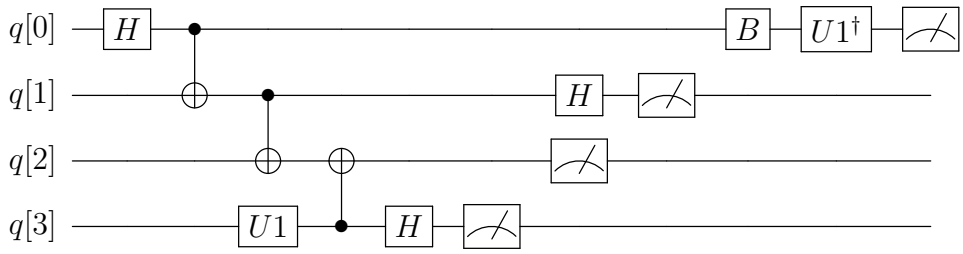

Figure 2.2 Controlled teleportation fidelity measurement circuit

As mentioned, the quantum computing platform API returns a probability distribution of measurement results after a batch of executions. An IBMQX4 API response contains a list of the outcomes that were measured as well as the number of executions in the batch that yielded each listed outcome as a percentage. The fidelity of a transmitted state was derived from an API response by summing the probabilities of all the measured outcomes in which both the observed Bell State and Charlie's result matched what was expected by the post-selection algorithm. This gave us the total probability of all the outcomes relevant to the post-selection. A second sum of the probabilities of the relevant outcomes in which Bob's resulting qubit also had a state of 0 indicated the fidelity of the transmission protocol. A ratio of this second sum over the total probability of the relevant outcomes provided the final, adjusted fidelity of the transmission being tested in a batch of executions. This process was first verified on IBM's simulator and was shown to have perfect fidelity when simulated. A subset of our data will provided in the data tables following. To make comparisons meaningful, the same subset will be presented in each tabular dataset. Therefore, we will

only show results for the input state $|\psi_2>$ in these tables. However, the complete datasets will be represented in more compact graphs.

Result Fidelity	Bell State	Input State
1	$\|\phi^+>_{xA}$	$\|\psi_2>$
1	$\|\phi^->_{xA}$	$\|\psi_2>$
1	$\|\psi^+>_{xA}$	$\|\psi_2>$
1	$\|\psi^->_{xA}$	$\|\psi_2>$

Table 2.3 Simulated results subset, allowed teleportation

To demonstrate the efficacy of Charlie's influence over the protocol, the procedure was done again, but the post-selection algorithm adjusted to not discriminate between Charlie's results. This yielded the following results.

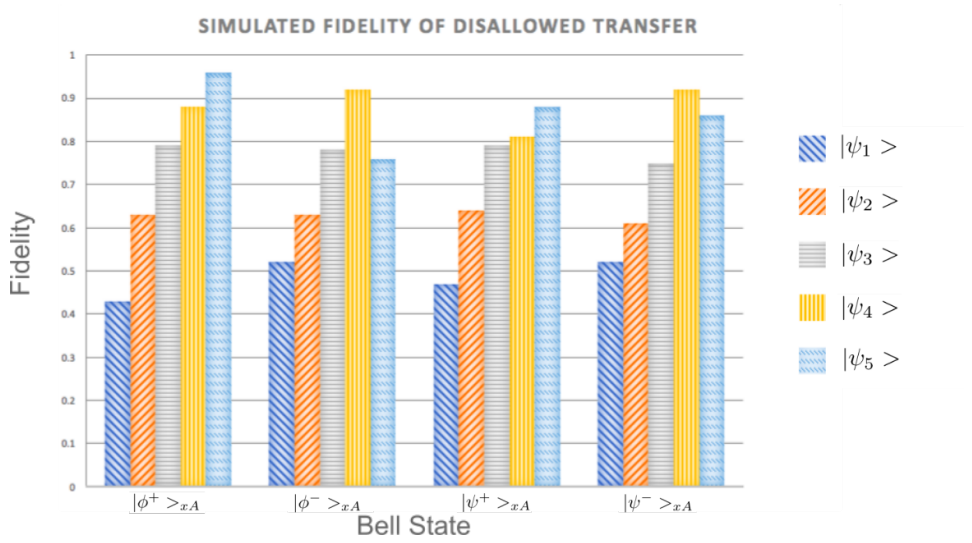

Figure 2.3 Fidelity of protocol across all simulated cases, disallowed teleportation

This test was successful as Charlie's influence on the protocol's fidelity was statistically significant. The protocol succeeded with 100 percent fidelity when Charlie allowed the

teleportation. In contrast, the disallowed teleportation fidelity had a mean of 71 percent and a mode of 76 percent.

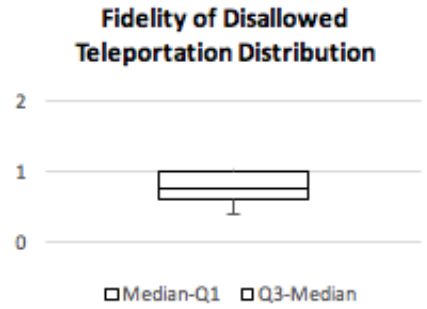

Figure 2.4 Distribution of fidelity across all simulated cases, disallowed teleportation

The next step was to implement the same procedures and tests that had been simulated on the quantum computer. These results show the protocol's performance in the real world.

Result Fidelity	Bell State	Input State
0.88	$\|\phi^+>_{xA}$	$\|\psi_2>$
0.92	$\|\phi^->_{xA}$	$\|\psi_2>$
0.85	$\|\psi^+>_{xA}$	$\|\psi_2>$
0.86	$\|\psi^->_{xA}$	$\|\psi_2>$

Table 2.4 Computed results subset, allowed teleportation

The procedure was done on the computer again with the post-selection algorithm adjusted to not discriminate between Charlie's results. This demonstrated that Charlie did not have the same level of control over the protocol as had been shown on the simulator.

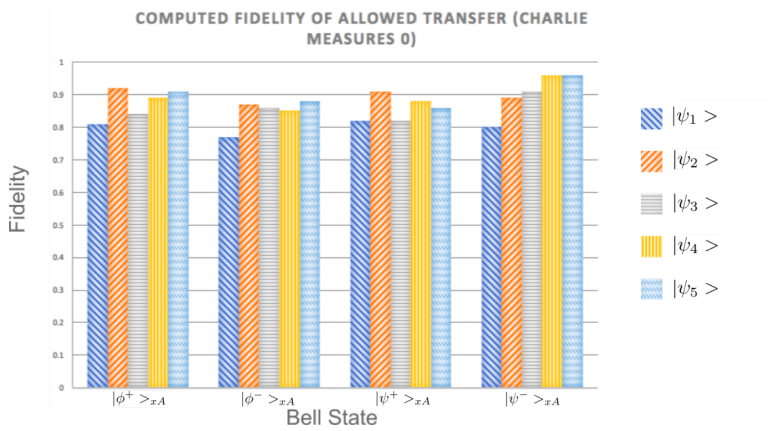

Figure 2.5 Fidelity of protocol across all computed cases where Charlie measures 0, allowed teleportation

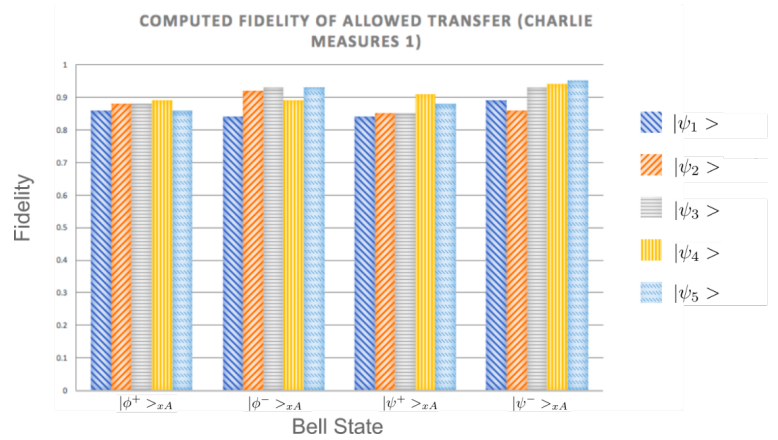

Figure 2.6 Fidelity of protocol across all computed cases where Charlie measures 1, allowed teleportation

Result Fidelity	Bell State	Input State
0.73	$\|\phi^+>_{xA}$	$\|\psi_2>$
0.63	$\|\phi^->_{xA}$	$\|\psi_2>$
0.54	$\|\psi^+>_{xA}$	$\|\psi_2>$
0.65	$\|\psi^->_{xA}$	$\|\psi_2>$

Table 2.5 Computed results subset, disallowed teleportation

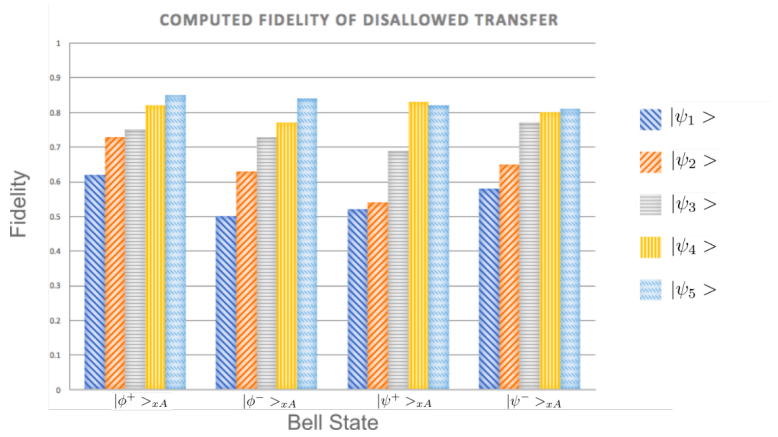

Figure 2.7 Fidelity of protocol across all computed cases, disallowed teleportation

On the computer, Charlie's influence on the protocol's fidelity was still evident, but less significant than in the simulated results. The protocol performed with a mean of 87 percent and a mode of 88 percent fidelity when Charlie allowed the teleportation. The disallowed teleportation fidelity had a mean of 71 percent and a mode of 74 percent.

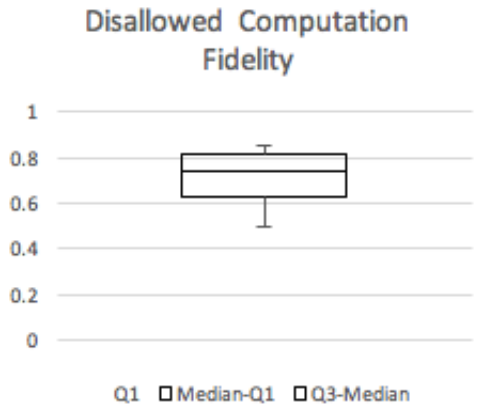

Figure 2.8 Distribution of fidelity across all computed cases, disallowed teleportation

A measure of statistical significance between two distributions is the difference between their medians, divided by the maximum range of the 1^{st} and 3^{rd} quartiles of both. Using

Figure 2.9 Distribution of fidelity across all computed cases, allowed teleportation

this measure, it can be shown that the influence of Charlie on the computed protocol has some significance. This is shown in the equation below with F representing fidelity, $Q3$ as the 3^{rd} quartile and $Q1$ as the 1^{st} quartile.

$$\frac{\overline{F}_{allowed} - \overline{F}_{disallowed}}{max(Q3_{allowed}) - min(Q1_{disallowed})} = \frac{0.87 - 0.71}{0.92 - 0.62} = 57\% \tag{2.7}$$

2.6 Superdense Coding

Superdense coding is a technique that leverages entanglement to achieve the communication of more information than would be classically possible with a number of bits physically transmitted between two parties.

This method is demonstrated in the quantum circuit diagram below with $q[1]$ representing Alice's entangled qubit and $q[0]$ representing Bob's entangled qubit.

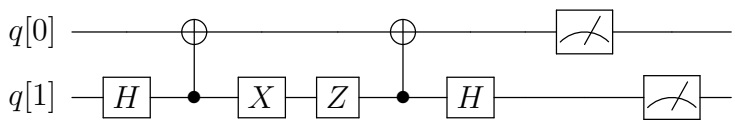

Figure 2.10 A simple superdense coding circuit

Initially, Alice entangles both qubits and gives one to Bob. This must occur before superdense communication takes place. Once Alice has a message to transmit to Bob, she then performs a combination of the X and Z gates to the qubit she still has in order to encode her message.

Message	Gates
00	I
01	Z
10	X
11	XZ

Table 2.6 Alice's superdense encoding operations

After performing these gates, Alice sends her qubit to Bob who will perform a and Hadamard to the qubits as shown in the circuit before measuring both. The result of his measurement will yield the message corresponding to Alice's encoding operation.

2.7 Controlled Dense Coding

A controlled dense coding scheme [HLG01] introduces a controlling party to the protocol, providing a similar mechanism of control to that which was seen in the controlled teleportation protocol. The implementation of this protocol can be represented by a quantum circuit with the gate labelled A representing Alice's encoding operation, $q[0]$ as the controller's qubit C, $q[1]$ as B and $q[2]$ as A.

The protocol begins with the preparation of a GHZ state. Each of Alice, Bob and Charlie then receive one of the entangled qubits. Charlie then performs a measurement in the x basis

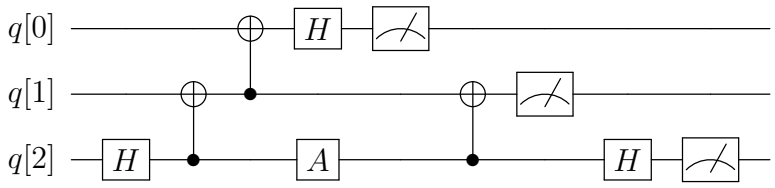

Figure 2.11 A controlled superdense coding circuit

of the C qubit. If Charlie measures a result of $|0>$ this represents the $|-x>$ eigenvector of the x basis and indicates that a transmission may be successful. Charlie may then classically communicate to Bob that the conditions for success are met and he can safely decode Alice's message. Charlie may also choose to disallow the operation by refraining from reporting the result of his message. To encode her message, Alice will perform one of the following gate operations.

Message	Gates
00	I
01	Z
10	X
11	Y

Table 2.7 Alice's controlled superdense encoding operations

The operation used to encode the message "11" is different from that used in the simple superdense coding algorithm because in this scheme, the two bits of information are encoded differently. Bob decodes Alice's message by determining the parity and phase of Alice's encoding operation.

To determine the parity of Alice's operation, Bob performs and measures qubit $CNOT_{AB}$. The result indicates whether Alice's message was of even parity or odd.

There are two possible odd parity and two possible even parity operators for Alice to

use. Therefore, once Bob has determined the parity of Alice's operation he can still decode another bit of information by determining the exact operation that Alice performed. Bob does this by measuring qubit B in the x basis, yielding either $|-x>$ or $|+x>$.

and narrowing down exactly what operation Alice performed. The results of these measurements correspond directly to the message encoded by Alice as shown. If Bob attempts to decode a message without knowledge of Charlie's result, his decoding operation will not necessarily indicate the correct phase and parity of Alice's operation and will not be reliable.

2.8 Implementation on the IBMQX4

In order to demonstrate the efficacy of Charlie's control in the controlled dense coding protocol defined in section 6, the protocol was implemented and tests of effectiveness automated to run on IBM's quantum computing platform. The protocol was first verified on IBM's simulator and was shown to have perfect fidelity when simulated. Each message was encoded and decoded 1000 times for allowed transmissions and 1000 times for disallowed transmissions. "Charlie's relevance" represents the percentage of a batch of 1000 executions for which the result of his measurement was $|-x>$ in an allowed transmission or falsely reported to be $|-x>$ in a disallowed transmission. The fidelity of a batch of 1000 executions is represented by the percentage of executions in which Bob decoded the correct message prepared by Alice.

The same protocol was repeated on the IBMQX4 to show the extent of Charlie's control in the real world.

Result Fidelity	Allowed	Message
1	YES	00
1	YES	01
1	YES	10
1	YES	11
0.49	NO	00
0.50	NO	01
0.52	NO	10
0.51	NO	11

Table 2.8 Simulated results, superdense coding

Result Fidelity	Allowed	Message
0.83	YES	00
0.84	YES	01
0.78	YES	10
0.78	YES	11
0.52	NO	00
0.49	NO	01
0.46	NO	10
0.46	NO	11

Table 2.9 Computed results, superdense coding

When the protocol was performed on the IBMQX4 it did not perform with perfect fidelity, but had an allowed success rate of 80.75 percent and a disallowed success rate of 48.25 percent.

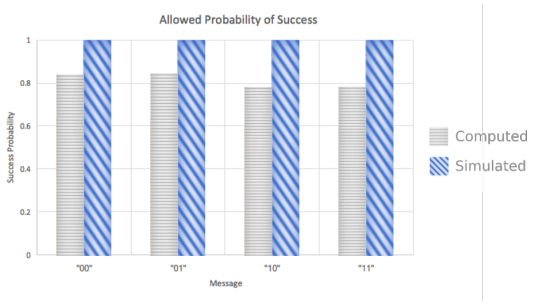

Figure 2.12 Fidelity of allowed messages

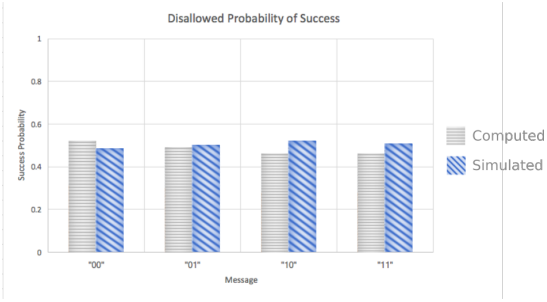

Figure 2.13 Fidelity of disallowed messages

2.9 Discussion and Summary

Quantum communication carries incredible implications for the future of information technology and security. A perfectly functioning communication system built on the principles on quantum teleportation could introduce the world to completely secure communication. As the possibility of such systems is being considered and working models are being demonstrated in simulations, it is important for us to plan ahead and to know the measure of control we will be able to exercise over communications. In this chapter it has been shown that models for a quantum communication channel with a remote controller who can allow or disallow communication is theoretically sound and performs perfectly when simulated.

Furthermore, it has been shown that the same models perform with decreased fidelity when executed on an actual quantum computer, the IBMQX4. The effectiveness of the remote controller has been shown to be recognizable overall. Despite decoherence effects due to gate and qubit errors which caused the experimental circuits to have an average

output fidelity of only 90%, the difference between the fidelities of allowed versus disallowed communications was shown to be statistically significant.

Some interesting results considered in isolation also tell their own stories. For example, the fidelity of the disallowed teleportation of the $|\phi^+\rangle_{xA}$ state is higher on the IBMQX4 than in simulations. It seems counterintuitive that a protocol may yield higher fidelity in an experimental setting than in a simulation. However, it is important to remember that the fidelity measured is that of the state that Alice is attempting to teleport. It is the intended outcome of the disallowed teleportation that the fidelity of this is low, which demonstrates Charlie's control. Therefore, the disallowed teleportation protocol is actually performing better in simulations, as expected. The success of the controlled protocols on the IBMQX4 are demonstrated by the statistical differences between the fidelities of disallowed versus allowed cases. To contrast the simulated and computed cases, we should compare these statistical differences.

We can calculate the statistical significance of the separation between the allowed and disallowed simulations in a similar way to how we calculated the same for the experimental computations in equation 2.7. The box-and-whisker graph for the disallowed teleportation distribution is provided. There is no distribution for the allowed case since the fidelities are all simply 1.

$$\frac{\overline{F}_{allowed} - \overline{F}_{disallowed}}{max(Q3_{allowed}) - min(Q1_{disallowed})} = \frac{1 - 0.77}{1 - 0.64} = 64\% \tag{2.8}$$

Since $64\% > 57\%$, the difference in allowed and disallowed distributions is much more significant in the simulations than in the experimental computations. This shows that there is still significant room for improvement of the experimental controlled teleportation protocol realization.

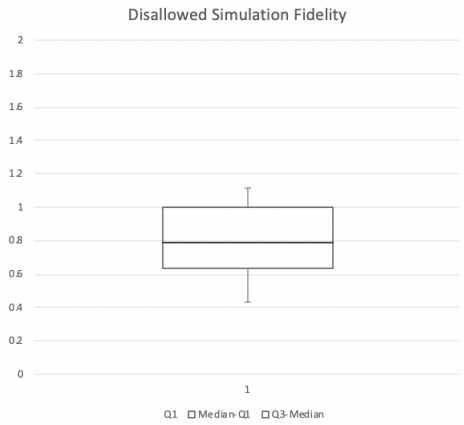

Figure 2.14 Distribution of fidelity across all simulated cases, disallowed teleportation

Box-and-whisker graphs are not as appealing to show for the controlled dense coding protocol since the allowed and disallowed fidelity distributions do not overlap at all in this protocol. However, we can still calculate the statistical significance of the simulated distributions' separation.

$$\frac{\overline{F}_{allowed} - \overline{F}_{disallowed}}{max(Q3_{allowed}) - min(Q1_{disallowed})} = \frac{1 - 0.505}{1 - 0.49} = 97\% \tag{2.9}$$

We can show that the statistical significance of the controller is less in the computed case.

$$\frac{\overline{F}_{allowed} - \overline{F}_{disallowed}}{max(Q3_{allowed}) - min(Q1_{disallowed})} = \frac{0.81 - 0.48}{0.84 - 0.46} = 87\% \tag{2.10}$$

In both the controlled teleportation and controlled dense coding protocols, the controller has higher statistical significance in simulations than in experimental computations. This shows that the experimental implementations of quantum computers for networked communication still have room for improvement, but that the protocols can also be demonstrated now with significant results.

This study has not considered how the communication protocols behave when the sending and receiving parties do not participate, since the purpose of this study was to determine the effectiveness of control. This could be an interesting topic for a future work. It would also be interesting for a future work to examine how and if the security of quantum communication protocols are affected by the participation of a controller in a networked communication setting.

Chapter Three

Quantum Blockchain

A Review of Quantum Blockchain Technology.

3.1 Chapter Overview

Blockchain technology is facing critical issues of scalability, efficiency and sustainability. These problems are necessary to solve if blockchain is to become a technology that can be used responsibly. Useful quantum computers could potentially be developed by the time that blockchain will be widely implemented for mission-critical work at financial and other institutions. Quantum computing will not only cause challenges for blockchain, but can also be harnessed to better implement parts of blockchain technologies including cryptocurrencies. We review the work that has been done in the area of quantum blockchain and hybrid quantum-classical blockchain technology and discuss open questions that remain.

3.2 Introduction

Quantum blockchain technology is one of the areas of research in the rapidly growing field of quantum cryptography [Feh10]. Quantum cryptographic schemes make use of quantum mechanics in their designs. This enables such schemes to rely on presumably unbreakable laws of physics for their security. Many quantum cryptography schemes are information-theoretically secure, meaning that their security is not based on any non-fundamental assumptions. In the design of blockchain systems, information-theoretic security is not proven. Rather, classical blockchain technology typically relies on security arguments that make assumptions about the limitations of attackers' resources.

Blockchain and distributed ledger technologies have applications in many industries, most notably in the financial industry. The financial applications of blockchain technologies include cryptocurrencies, insurance and securities issuance, trading and selling. Non-financial applications of blockchain technology have been identified for the music industry, decentralized IoT, anti-counterfeit solutions, internet applications and decentralized storage, to name a few. In recent years, blockchain projects have attracted massive attention in these

industries [Nof+17].

Despite being a relatively new technology, blockchain has made significant waves in a number of important industries in a very short time. The two most known instances of blockchain technologies are Bitcoin [Nak] and Ethereum [But13], which are the core of modern cryptocurrencies. Ethereum's focus on smart contracts has made it a valuable tool for decentralizing numerous industries.

The philosophical implications of decentralized consensus technologies are far-reaching. Atzori suggested in 2015 that all of society might be restructured by the blockchain, and that "the decentralization of government services through permissioned blockchains is possible and desirable" [Atz15].

In this chapter we review work that introduces quantum cryptographic methods to blockchain technology. We discuss the potential impact and risk associated with blockchain technology and how the proposed quantum cryptographic methods attempt to address these risks.

3.3 Blockchain Background

Before delving into quantum and hybrid quantum-classical blockchain cryptography schemes, we will provide a brief summary of the core mechanisms in blockchain technology. The National Institute for Standards and Technology (NIST) describes blockchain technology in the following way:

> Blockchains are tamper-evident and tamper-resistant digital ledgers implemented in a distributed fashion (i.e., without a central repository) and usually without a central authority (i.e., a bank, company, or government). [Yag+18]

We will focus primarily on Ethereum's blockchain implementation in the following sections since Ethereum's smart contracts have inspired interesting work in theoretical

quantum blockchain design. Ethereum was introduced by Vitalik Buterin in 2013 [But13]. The most important feature of Ethereum is arguably its Turing-complete scripting language for smart contracts. Ethereum shares many basics with other blockchain implementations. Here, we will summarize the elements of Ethereum that are most relevant to the work that we review [But13]. We begin with the basics that are shared by Ethereum and Bitcoin.

3.3.1 The Ledger

The distributed ledger of a blockchain cryptocurrency maintains the ownership and status of all existing coins. The ledger is made up of a chain of blocks. The chain is composed of the blocks' references to one another. Any valid block's header contains a hash of the header of the previous block in the chain. Each block typically also contains a timestamp, nonce, and list of transactions.

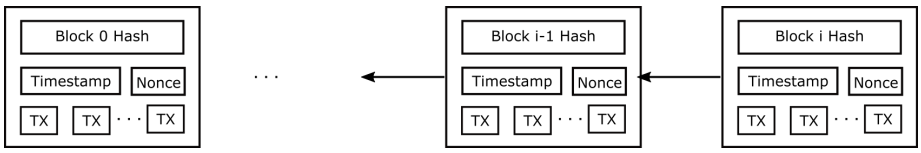

Figure 3.1 Chain of Blocks

When a transaction occurs, the current ledger state is mutated by a function that takes the original state S_0 and the transaction TX, and outputs the next state S_1 or an error E. Here, and throughout this chapter, $\leftarrow$ represents a transition of a state. Note that in this case, the state is a purely classical data structure. However, in later sections the same notation will be used to denote transitions between quantum states.

$$S_1 or E \leftarrow Apply(S_0, TX) \tag{3.1}$$

In Bitcoin, a ledger's state is composed of all Unspent Transaction Outputs ($UTXO$), or simply all of the coins that have been mined but not spent. Each coin has a 20-byte cryptographic public key which contains information about its owner and its denomination.

A transaction requires references to each $UTXO$ involved and the cryptographic signatures produced by the $UTXO$ owners' private keys.

3.3.2 Proof of Work

To achieve the decentralization of the ledger, a consensus system must be introduced. The goal of the consensus system is to ensure that everyone agrees on the validity of the transactions that have led to the ledger's state and their order. There are several consensus systems that are in use today, including proof-of-work, proof-of-stake, proof-of-burn and more. The most ubiquitous is proof-of-work.

Bitcoin's proof-of-work based system requires that users attempt to publish transactions constantly. These transactions are published in packaged groups of a fixed size (1 MB in the case of Bitcoin) called blocks. In addition to a list of transactions, a block contains a timestamp, one-time use block id or nonce, and a hash of the header of the last most-recent block that contributed to the ledger. Hence each block maintains a reference to the block that came before it, and the blocks form a chain as they are published which reflects the order of their publications in time.

In order for a block to be accepted, its proof of work must be valid. The validity condition for Bitcoin block is that its double-SHA256 hash is less than a dynamically adjusted cutoff when interpreted as an integer. A SHA256 hash is a completely unpredictable result of a pseudorandom function. So, in order to create a valid block the hash function must be run an arbitrary number of times until a valid output randomly occurs. Therein lies the work that must be done to generate a proof-of-work, and the incredible overhead in computational resources that is encouraged by proof-of-work blockchains.

The time required to generate a valid hash is fundamental to the consensus system that is employed. If an attacker attempts to move money in a way that conflicts with the ownership of coins as a result of a transaction record already accepted by the ledger, the attack is simply rejected. However, an attacker can try to fabricate a block which points to a valid block that was published *before* the block containing the transaction which changed the ownership of the desired coins. In this case, the attacker will be required to generate a new valid proof-of-work. While the attacker is occupied doing this, it is assumed that many other miners are continuing to publish blocks that point to the latest legitimate block. The rule that is applied to weed out these attacks is simply that the longest valid chain is taken to be the truth. An attacker would therefore need to have more computing power than the rest of the network combined in order to outpace the speed of the network's publications and make his/her chain the longest. This is called a 51% attack and would theoretically be successful.

3.3.3 Proof of Stake

Proof-of-stake schemes were introduced to address some of the issues with proof-of-work [Fra20]. Ethereum is currently in the process of switching to a proof-of-stake scheme. The mining power available to a miner in a proof-of-stake scheme is proportional to the number of coins owned by the miner. Hence, they are limited to mining a number of blocks that is proportional to their stake in the cryptocurrency ecosystem. This offloads the miners' consumption of electrical energy resources to currency resources that are more internal to the blockchain.

A driving force behind the creation of proof-of-stake was the dynamic created by miners selling their coins to pay off their electrical bills. This movement of cryptocurrency out of the ecosystem has led to drops in cryptocurrency value.

Proof-of-stake schemes have less inherent risk than proof-of-work schemes. This is clearly illustrated by the proof-of-stake version of the 51% attack. In a proof-of-stake blockchain, an

attacker would need to have 51% of the cryptocurrency in the ecosystem to make a successful 51% attack. This would make it unappealing to attack the ecosystem, since destroying the security and validity of the system would risk invalidating the attacker's virtual fortune. This is a natural deterrent that does not exist in proof-of-work schemes, where any attacker with 51% of the network's computing power can make a successful 51% attack regardless of their stake in the ecosystem.

There are cons to any consensus algorithm. In the case of proof-of-stake, one problem is the explicit association of wealth with the power to influence events. While the scheme improves on proof-of-work in some ways, it still incentivizes competition and, similarly to evolutionary systems, rewards the "fittest" competitor. In this case, fitness is quantified by units of currency rather than computational capabilities.

3.3.4 Smart Contracts

One of Ethereum's most significant contributions to blockchain technology is the concept of autonomous smart contracts. The addition of smart contracts differentiates so-called "Blockchain 2.0" technology like Ethereum from "Blockchain 1.0" technology like Bitcoin [Edi19]. Blockchain 2.0 technologies enable programmers to use autonomous agents, the smart contracts, as elements of distributed software applications called Distributed Applications (DApps).

The top-level data structures in Ethereum's ledger are accounts, rather than coins. The ledger maintains each account's 20-byte public-key address, nonce, balance, contract code and storage.

There are two types of accounts. The first is Externally Owned Accounts, which are controlled by private keys. The second is Contract Accounts which are controlled by their contract codes. Externally owned accounts are similar to those used by Bitcoin, and can be used in transactions as described previously. Contract accounts are much more interesting.

Contract accounts act as autonomous agents which execute their contract code when sent messages. A contract account can be programmed to automatically read and write to its storage, send additional messages to other contract accounts, or create transactions. To cause a contract account to execute its code exacts a monetary price on the sender of the original message. This price is known as "gas" and is proportional to the complexity of the contract code. The money (Ether) provided in the original message is used as gas to "fuel" all contract code executions that result from the first contract's activation.

Contract code is written in a low-level stack machine-based bytecode language called Ethereum Virtual Machine (EVM) code. The language makes use of a stack, linear memory array and long term storage. The language is composed of a small instruction set that includes blockchain application-specific instructions like $CALL$ which sends a message to a contract and $CREATE$ which creates a new contract.

The blocks used by Ethereum are very similar to those used by Bitcoin. Ethereum blocks contain all the information that a Bitcoin block does, with the addition of a copy of the most recent ledger state, the block number and a record of the mining difficulty for that block. Ethereum does not fundamentally deviate from the typical proof-of-work consensus scheme.

3.4 Quantum Coins

A straightforward way to introduce quantum technology to the blockchain at the cryptocurrency level is to simply reference the many schemes for quantum money that have been defined since 1960 [Wie83]. Bitcoin and Ether were described in section 3 as the representations of monetary value that are traded between parties through transactions. These coins have monetary value and cryptographically protected ownership records. Coins are one of the primitive data structures required to formulate a cryptocurrency blockchain.

3.4.1 Public-Key Quantum Money

In the case of public-key quantum money, the scheme takes advantage of superposition to ensure that when a quantum state is used as a coin no bad actor can duplicate the coin. An attacker cannot know which basis to measure each qubit of the quantum state in without knowing the secret key which was originally used to create the state. The attacker cannot learn the state without performing the correct measurement due to the no-cloning restriction.

The procedure to generate public-key quantum money is very straightforward, and was originally introduced in 1960 by Stephen Wiesner [Wie83]. This paper arguably kicked off the field of quantum cryptography and directly inspired the design of BB84 quantum key distribution [BB87].

An algorithm for public-key quantum money generation is simply the following:

1. Generate two random bit strings M and N of length l
2. Prepare a quantum state $|\$ >= |0>^{\otimes l}$
3. For each bit $i < l$:
   - If $M_i = 0$ and $N_i = 0$, do nothing to the i^{th} qubit
   - If $M_i = 0$ and $N_i = 1$, rotate the i^{th} qubit state to $|1>$
   - If $M_i = 1$ and $N_i = 0$, rotate the i^{th} qubit state to $|+>$
   - If $M_i = 1$ and $N_i = 1$, rotate the i^{th} qubit state to $|->$

M_i and N_i are kept secret by the mint, and $|\$>$ is published as the quantum public key. In this case, only the mint has enough knowledge to verify the public key and no one can duplicate it.

3.4.2 Binding Commitments

Some of the mechanisms that were taken for granted in the description of classical blockchain technology are non-trivial to implement using quantum algorithms. For example, we described in section 3.1 that a transaction requires references to each $UTXO$ involved and the cryptographic signatures produced by the $UTXO$ owners' private keys. This information is necessary to validate the ownership of the coins involved in the transaction. Using his/her private key, the owner of the coins creates a digital signature so that other parties can verify that the transaction was indeed authorized by the owner of the private key and was not modified since. Using the corresponding public key and the signature, any party can verify the validity of the transaction without learning the private key. This is the basic premise of public-key cryptography.

A blockchain transaction using quantum money will still require references to each $UTXO$ involved and the cryptographic signatures produced by the $UTXO$ owners' private keys in order to verify ownership. It makes sense to also require that a user who has committed coins to a transaction in good faith *must* produce his/her signature when it is time for the transaction to be approved. This would make the creation of a transaction using quantum public-key money as coins a type of binding commitment.

Computationally binding commitment schemes between two parties are composed of two phases. The Commitment Phase allows one party to send the other party some information c related to a message m which does not give the receiver any information about m itself. However, the act of sending c binds the sender to provide the message m in the second stage, the Open Phase. In the Open Phase, the sender transmits m to the receiver and proves to the receiver that m does indeed correspond to c by providing a signature that "opens c to m".

A classical definition of a computationally binding is the following from Unruh [Unr16b].

Definition 1 (Classical-style binding) *No algorithm A can output a commitment c*

and two signatures s, s' that open c to two different messages m and m'.

Computationally binding commitment schemes have been studied and defined in the quantum setting [Unr16b; ARU14; Feh18; Unr16a]. Interestingly, when the algorithm A is allowed to be a quantum polynomial-time algorithm, this definition was shown to be inadequate. While definition 1 holds for a particular classical-style binding commitment, Ambainis, Rosmanis, and Unruh showed that for this particular binding a quantum polynomial-time algorithm A employed by an adversary could open c to any message that the adversary wished [ARU14]. Therefore Unruh was motivated to define a different type of binding that was useful in the quantum case. The new binding property is demonstrated by a pair of quantum games.

Let A, B be algorithms and S, M, U be quantum registers. V_c is a measurement which verifies that that U opens M. M_{ok} measures m in the computational basis if $ok = 1$.

The first game $Game_1$ consists of four steps:

$$(S, M, U, c) \leftarrow A(1^\gamma) \tag{3.2}$$

$$ok \leftarrow V_c(M, U) \tag{3.3}$$

$$m \leftarrow M_{ok}(M) \tag{3.4}$$

$$b \leftarrow B(1^\gamma, S, M, U) \tag{3.5}$$

The second game $Game_2$ omits the measurement in step three but is otherwise the same:

$$(S, M, U, c) \leftarrow A(1^\gamma) \tag{3.6}$$

$$ok \leftarrow V_c(M, U) \tag{3.7}$$

$$b \leftarrow B(1^\gamma, S, M, U) \tag{3.8}$$

A commitment scheme is "collapse-binding" iff for any quantum polynomial time valid adversary, $cAdv = |Pr[b = 1 : Game_1] - Pr[b = 1 : Game_2]|$ is negligible.

This essentially expresses that if an adversary (A, B) provides a classical commitment c, there must be only one message they can open c to. A outputs a superposition of messages M and a superposition of corresponding opening signatures U. S is the adversary's state. The assertion that $|Pr[b = 1 : Game_1] - Pr[b = 1 : Game_2]|$ is negligible limits the value of M to computational basis vectors for collapse-binding commitments. No quantum polynomial-time algorithm B should be able to distinguish between the value of M whether M is measured in the computational basis or not.

3.4.3 Collapsing Hash Functions

The games used to define the collapse-binding property of commitment schemes can also be applied to classify hash functions that are collapsing [Unr16b]. Assume H is a one-to-one hash function.

Definition 2 (Collapsing hash function - informal) *H is a collapsing hash function iff no quantum polynomial time algorithm B can distinguish between $Game_1$ and $Game_2$. An adversary is valid if A outputs a classical value c and a register M where $H(m) = c$.*

This game-based definition was clarified and made mathematical by Fehr in 2018 [Feh18].

Definition 3 (Collapsing hash function - formal) *A function $H\ \mathbb{X} \to \mathbb{Y}$ is $\in(q)$-collapsing if*

$$cAdv[H](q) := \sup_{SMCU} \delta_q(M, \overline{M}|\overline{C}U) \leq \in (q) \tag{3.9}$$

for all q. The supremum is over all states $SMCU = S\ H(M)\ CU$ with complexity $\leq q$.

The collapsing property of a hash function is a counterpart of collision resistance. Unruh shows that the random quantum oracle is a collapsing hash [Unr16b] and so some hash function based commitment schemes are collapsing in the random oracle model. Unruh also showed that Merkle-Damgard hash functions are collapsing if their underlying compression algorithms are, which implies that SHA-2 is collapsing [Unr16a]. Czajkowski, et al. showed the same for Sponge hashes with certain conditions [Ber+08]. Sponge hash construction underlies SHA-3.

3.4.4 Collision Free Quantum Money

Collision free quantum money is a concept that was introduced by Lutomirski, et al. [Lut+09]. The premise is that a mint can not efficiently produce two coins with the same verification circuit, and so each coin made is unique. This is a step towards remedying the problem with Wiesner's public-key quantum money. In Wiesner's scheme, only the mint can verify the quantum public keys of minted coins. This is an important issue specifically in the context of blockchain, since the intention is specifically *not* to have a centralized signing authority in a distributed system.

Let L be a classical function that assigns a unique label to each exponentially small subset of a superset of elements. L should also be as obscure and unstructured as possible. The procedure for generating collision-free quantum money is the following.

- Begin with an equal superposition over all n-bit strings.
- Compute L into an ancilla register and measure that register to obtain a value l.

This procedure would have to be repeated exponentially many times to produce the same value l twice. The quantum state will then be $|\$_l>$, an equal superposition of exponentially many terms with no clear relationship to one another.

$$|\$_l >= \frac{1}{\sqrt{N_l}} \sum_{xs.t.L(x)=l} |x > \tag{3.10}$$

Verification can be done using rapidly mixing Markov chains. Verification requires knowledge of a Markov matrix M that will rapidly mix from any distribution over bit strings with the same l to the uniform distribution of those same strings. No string with a different l can be present in that final uniform distribution. Each update consists of a uniform random choice over N update rules P_i. Each update rule is deterministic and invertible. Then any valid quantum money state will be a $+1$ eigenstate of M.

$$M^r \doteq \sum_l |\$_l >< \$_l| \tag{3.11}$$

$$M = \frac{1}{N} \sum_{i=1}^{N} P_i \tag{3.12}$$

The verification procedure makes use of a unitary U.

$$U = \sum_i P_i \otimes |i >< i| \tag{3.13}$$

The verification procedure itself is the following:

- Introduce an ancilla in uniform superposition over all i.
- apply U.
- Measure the projector of the ancilla onto the uniform superposition.
- discard the ancilla.

The outcome 1 has a corresponding Kraus operator sum element:

$$(I \otimes \frac{1}{\sqrt{N}} \sum_{i=1}^{N} < i|)U(I \otimes \frac{1}{\sqrt{N}} \sum_{i=1}^{N} |i >)$$

$$= \frac{1}{N} \sum_{i=1}^{N} P_i$$

$$= M \tag{3.14}$$

Repeating the procedure r times brings the Kraus operator to M^r, and achieves an approximation of a measurement of $\sum_l |\$_l >< \$_l|$.

3.4.5 Quantum Lightning

A recent construction of quantum money is Quantum Lightning, which was proposed by Zhandry in 2017 [Zha19]. Quantum Lightning is a formalization of collision-free quantum money [Lut+09].

Quantum Lightning makes use of non-collapsing collision-resistant hash functions. These hash functions are defined by a random set of degree-2 polynomials over $\mathbb{F}_2$. Quantum Lightning defines the "Lightning Bolt" state $|\lightning >$. The verification procedure **Ver** for bolts is another polynomial-time quantum algorithm that either outputs the serial number of a valid bolt, or $\perp$ for invalid bolts. The serial number of a bolt is a deterministic function of the bolt itself, and verification does not perturb the bolt. Bolts are created by quantum algorithms called "Storms" and denoted ⛈.

A bolt is generated by the following procedure.

1. Randomly choose n random upper-triangular matrices $A_i \in \{0,1\}^{m \times m}$, and set $\mathbb{A} = \{A_i\}_i$. $\mathbb{A}$ is the public key. Let the hash function $f_{\mathbb{A}} : \{0,1\}^m \to \{0,1\}^n$ be $f_{\mathbb{A}}(x) = (x^T \cdot A_i \cdot x)_i$. If we let operations be taken mod 2, this captures general degree 2 functions over $\mathbb{F}_2$.

2. Begin with a state $|\phi_0>$.

$$|\phi_0>=$$
$$\frac{1}{2^{kn/2}}\sum_{\Delta_1,...,\Delta_k}|\Delta_1,...,\Delta_k> \tag{3.15}$$

3. Δ is defined such that we can run a computation which maps $\Delta = (\Delta_1,...,\Delta_k)$ to an affine space S_Δ s.t. $\forall x \in S, f_{\mathbb{A}}(x) = f_{\mathbb{A}}(x+\Delta_j)\forall j$. Then we construct a uniform superposition of elements in S_Δ to yield:

$$|\phi_1>= \tag{3.16}$$
$$\sum_{\Delta}\sum_{x\in S_\Delta}\frac{1}{2^{kn/2}\sqrt{|S_\Delta|}}|\Delta,x> \tag{3.17}$$

4. Compute $f_{\mathbb{A}}$ in superposition and measure the resulting serial number y.

$$|\phi_y>\propto$$
$$\sum_{\Delta,x\in S_\Delta:f_{\mathbb{A}}(x)=y}\frac{1}{|S_\Delta|}|x,\Delta> \tag{3.18}$$

5. Compute the maps $(x,\Delta_1,...,\Delta_k)$ to $(x,x-\Delta_1,...,x-\Delta_k)$ in superposition. The final state is a bolt:

$$|\text{↯}_y>\propto$$
$$\sum_{\Delta,x\in S_\Delta:f_{\mathbb{A}}(x)=y}\frac{1}{|S_\Delta|}$$
$$|x,x-\Delta_1,...,x-\Delta_k>$$

$$\doteq \sum_{x_0,...,x_k : f_{\mathbb{A}}(x_i)=y \forall i} |x_0, ..., x_k >$$
$$= (\sum_{x : f_{\mathbb{A}}(x)=y} |x >)^{\otimes(k+1)}$$
$$= |\lightning'_y >^{\otimes(k+1)} \tag{3.19}$$

To verify a bolt, each of $k+1$ sets of the m registers is verified individually. Each of these "mini verifications" yields either an element in $\{0,1\}^n$ or $\perp$. Each mini verification must agree, and have the same output for the bolt to be valid.

We assume the mini verification is given $|\phi >= |\lightning'_y >$ that corresponds to some serial number y. The first step of mini verification is to check if the input state $|\phi >$ is in the space spanned by $|\lightning'_z >$ as z varies. The second step is to evaluate $f_{\mathbb{A}}$ in superposition in order to learn which of the orthogonal $|\lightning'_z >$ states we have. Then, we can measure the result to obtain y. For the correct $|\phi >= |\lightning'_y >$ this does not perturb the state. This a useful property since it means that a bolt can theoretically be re-used.

Quantum Lightning ensures that any bolt generated by an honest mint is accepted with probability negligibly close to 1. It also ensures that no adversarial bolt generator can generate two coins with the same serial number which would both pass verification. Zhandry shows in [Zha19] that Quantum Lightning is secure under some assumptions of the multi-collision resistance of a degree-2 hash function. Zhandry also proved that *any* non-collapsing hash function can be used to construct Quantum Lightning, though there are currently no such known hash functions that are proven to be non-collapsing [Zha19].

3.5 A Hybrid Payment System

In February 2019, Coladangelo proposed a payment system based on Quantum Lightning [Col19]. Quantum Lightning guarantees that no generation procedure can easily create two coins with the same serial number, and no one can clone existing coins. However, the Quantum Lightning scheme itself does not include a mechanism for regulating the generation

of valid coins. This mechanism is introduced as a part of Coladangelo's hybrid blockchain payment system.

This payment system is the first use of smart contracts in a quantum setting. Any party can deposit a coin to a smart contract, setting that contract's serial number to match the coin's. The classical certificate that can be found by measuring a valid bolt can also be submitted to a smart contract. If a certificate submitted by a user corresponds to the serial number stored in the smart contract, this means that the user owns bolt. The contract releases all of its coins to the user.

Coladangelo's payment system also considers one of the challenges with practical quantum computing: state decoherence. The downside of using quantum states as coins are that these coins can't be reliably stored for any significant period of time. The payment system makes use of smart contracts to implement a mechanism for lost coin recovery. A user can send a message to a smart contract with a coin whose serial number is the serial number of a coin they have lost. Other users have a time window in which they can challenge this claim by demonstrating that they in fact own the coin with the submitted serial number. If a claim is not challenged, then the coins submitted to the smart contract are returned to the sender of the message, and the serial number of the contract is updated to that of the lost coin.

3.5.1 Classical Blockchain

The Global Ledger

The payment system is primarily a classical blockchain, but uses Quantum Lightning as its coins. The classical serial numbers and certificates of the quantum coins are the interface between the quantum and classical elements of the system. The classical blockchain uses a global ledger. The global ledger maintains three sets and the current time:

```
parties = {}
contracts = {}
allTransactions = {}
t = 0
```

Ledger State

The messages that the global ledger can handle are the following. These are each slightly modified from those given by Coladangelo for clarity and consistency of notation.

Register (id, num_coins) → (pid) allows a user to set their *id* and retrieve their *pid*, which addresses their data in the system. This constitutes the registration of a user with the system. This message can also include a number of coins, which will be set on the registered party's data structure.

Retrieve Party (pid) → (id, num_coins) can be used to request a registered party's information.

Pay (pid, pid', num_coins) → (trid) allows user *pid* to send coins to user *pid'*. If *pid* or *pid'* are not valid, simply return $\perp$. If *pid', pid* $\in$ *parties* and **pid.coins* $>$ *num_coins*, then:

$$*pid'.coins \leftarrow *pid'.coins + num\_coins \tag{3.20}$$

$$*pid.coins \leftarrow *pid.coins - num\_coins \tag{3.21}$$

$$trid \leftarrow |allTransactions| + 1 \tag{3.22}$$

$$allTransactions[trid] \leftarrow (pid, pid', num\_coins, time) \tag{3.23}$$

Retrieve Transaction (trid) $\rightarrow$ (allTransactions[trid]) allows users to retrieve transaction details.

Smart Contract $(\mathbf{pids}, \{(\mathbf{pid}, \mathbf{num\_coins_{pid}}) : \mathbf{pid} \in \mathbf{pids}\}, \mathbf{circuit}, \mathbf{st_0}) \rightarrow (\mathbf{cid})$ allows a user to create a contract. $\{(pid, num\_coins_{pid}) : pid \in pids\}$ are initial deposits for each user *pid*. If $pids \subseteq parties$, then a new contract can be created.

Retrieve Smart Contract (cid) $\rightarrow$ (params, coins) allows a user to retrieve the details of a contract if $cid \in contracts$. Otherwise, returns $\perp$.

Smart Contracts

The global ledger handles contract creation through the *Smart Contract* message. However, the contracts themselves handle the most functional contract-related messages.

The contract creation procedure is the following.

$$cid \leftarrow |contracts| + 1 \tag{3.24}$$

$$*cid.params \leftarrow (pids, \{(pid, num\_coins_{pid}) : pid \in pids\}, circuit, st_0) \tag{3.25}$$

$$*cid.num\_coins \leftarrow 0 \tag{3.26}$$

$$contracts[cid] \leftarrow *cid \tag{3.27}$$

Once created, the contract waits for an *Initialize with Coins* message to come from each user $pid \in *cid.params.pids$. If $*pid.coins \geq num\_coins_{pid} \; \forall \; pid \in *cid.params.pids$, then the following occurs.

$$*pid.coins \leftarrow *pid.coins - num\_coins_{pid} \; \forall \; pid \in *cid.params.pids \tag{3.28}$$

$$*cid.coins \leftarrow *cid.coins + num\_coins_{pid} \; \forall \; pid \in *cid.params.pids \tag{3.29}$$

$$st \leftarrow st_0 \tag{3.30}$$

The smart contract then enters the "execution phase": a loop which repeats until termination. The contract waits for a *Trigger* message from any user $pid \in parties$. This message will provide variables $(pid, witness, time, st, num\_coins)$. If $circuit(pid, witness, time, st, num\_coins) \neq \perp$, then the following occurs.

$$*pid.coins \leftarrow *pid.coins - num\_coins \tag{3.31}$$

$$*cid.coins \leftarrow *cid.coins + num\_coins \tag{3.32}$$

$$(st, result) \leftarrow circuit(pid, witness, time, st, num\_coins) \tag{3.33}$$

The *result* will indicate how many coins the smart contract should release to user *pid*.

$$*pid.coins \leftarrow *pid.coins + num\_coins \tag{3.34}$$

$$*cid.coins \leftarrow *cid.coins - num\_coins \tag{3.35}$$

Initialize with Coins (pid, cid, num_coins) → () allows a user to deposit coins into a contract. This is necessary for a contract to enter its execution phase. If $cid \notin contracts$ or $pid \notin *cid.params.pids$, returns $\perp$. Otherwise, the following occurs.

$$*cid.coins \leftarrow num\_coins \tag{3.36}$$

$$*pid.coins \leftarrow num\_coins \tag{3.37}$$

Trigger (pid, cid, witness, time, st, num_coins) → (result) allows a user to run the circuit associated with a contract with the given parameters. If $cid \notin contracts$, returns $\perp$. *pid* may be any element of *parties*.

3.5.2 Quantum Lightning Payments

In this hybrid blockchain scheme, we have not yet mentioned any use of quantum physics. Indeed, the ledger and contracts in Coladangelo's scheme are completely classical. The only element of the system which makes use of quantum effects is the payments system, which is uses Quantum Lightning as a primitive. Coladangelo's payment system defines five procedures:

- generate valid quantum coins
- make a payment
- file a claim for lost coins
- prevent malicious attempts at filing claims
- trade valid quantum coins for classical coins

Generating Valid Quantum Coins

The procedure uses the Quantum Lightning Bolt generation procedure ☁ and bolt verification procedure **Ver** both defined by Zhandry [Zha19], which are included in section 4.5.

$$|\text{⚡}\rangle \leftarrow \text{☁} \tag{3.38}$$

$$serial \leftarrow \mathbf{Ver}(|\text{⚡}\rangle) \tag{3.39}$$

Then to use created coins with the blockchain, the *Smart Contract* message may be sent to the global ledger to create a contract. Once this message has been processed, an *Initialize with Coins* message is also sent to the ledger with the *cid* matching the contract created by the *Smart Contract* message.

Making a Payment

A payment involves two parties, the payer P and payee P'. The payment procedure involves the following steps.

- P sends $|\text{⚡}>, cid, serial$ and num_coins to P'.
- P' sends a *Retrieve Contract* message to the ledger, retrieving the contract cid.
- P' accepts the payment if $cid \in contracts$ and $\mathbf{Ver}(|\text{⚡}>) = serial$.

Recovering Lost Coins

In order to recover lost coins, a user P uses the *Trigger* message to cause a smart contract to execute a circuit *BanknoteLost*. This circuit records a request at the current time to $*cid.state$, indicating that a *BanknoteLost* message began to be processed at this time. With the *Trigger* message, user P also provides the serial number $serial$ of the lost coin, and deposits a number of coins num_coins into the contract cid.

During the time t_{tr} that follows, another user P' has the chance to challenge the claim made by P by demonstrating true ownership of the coin with the serial number $serial$. Recall that a bolt of Quantum Lightning is generated using degree-2 polynomial hash function H. If P' has access to the bolt, they can verify it through a verification procedure A which will identify only their one, unique bolt and yield some $m \in \{0,1\}^\lambda$ such that $H(m) = serial$.

To challenge the claim made by P, P' can perform the following new bolt generation and verification:

$$m \leftarrow A(|\text{⚡}>) \tag{3.40}$$

$$|\text{⚡}> \leftarrow \text{☁} \tag{3.41}$$

$$serial' \leftarrow \mathbf{Ver}(|\text{⚡}>) \tag{3.42}$$

Then, P' sends a *Trigger* message to the contract with m and $serial'$, running a circuit *ChallengeClaim* which causes the lost coin recovery record to be erased from the contract's state and the coins deposited by P to be returned.

If a claim made by P goes unchallenged for time t_{tr}, P can perform the bolt generation and verification procedure and send a *Trigger* message to the contract with the new coin's serial number $serial'$, which will run a circuit *ClaimUnchallenged*. This circuit simply updates the contract's serial number to be the new bolt's and removes the record of the recovery request.

Trading a Valid Quantum Coin for a Classical Coin

If a user P owns a quantum coin and wishes to redeem it for classical coins, they can demonstrate ownership by performing $m \leftarrow A(|\text{ϟ}>)$, and then sending a *Trigger* message which contains m and runs a circuit *RecoverCoins*. The *RecoverCoins* circuit releases the coins which were originally deposited in the contract by P back to P.

3.6 A Quantum Blockchain Voting Protocol

Sun, Xin, et al. presented a protocol for voting on a quantum blockchain in January 2019 [Sun+18]. Voting can be a suitable application of blockchain technology since the blockchain makes it difficult for participants to falsify claims. Sun, Xin, et al. make use of quantum commitments to design a self-tallying voting protocol.

3.6.1 Voting Using Binding Commitments

The protocol is very simple. Like the other commitment schemes discussed in this chapter, the voting protocol involves two phases. These phases are called the "ballot commitment" and "ballot tallying" phases.

The steps to the ballot commitment phase are the following.

1. For each $i \in \{1, ..., n\}$ voter V_i generates the i^{th} row of an $n \times n$ matrix of integers $r_{i,1}, ..., r_{i,n}$ such that $\sum_j r_{io,j} = 0(mod\ n+1)$.

2. For each i, j voter V_i sends $r_{i,j}$ to voter V_j via a quantum secure communication.

3. Then each voter V_i knows the i^{th} column $r_{1,i}, ..., r_{n,i}$. V_i computes his/her masked ballot $\hat{v_i} = v_i + \sum_j r_{j,i}(mod\ n+1)$. V_i commits $\hat{v_i}$ to every tallier of the blockchain via a quantum commitment protocol.

Ballots are tallied by the following decommitment procedure. $v_i = 0$ is considered a disagreement with the proposal being voted on, $v_i = 1$ is considered an agreement.

1. Each voter V_i reveals $\hat{v_i}$ to every tallier of the blockchain by opening his/her commitment.

2. The talliers each run the Quantum Honest Success Byzantine Agreement Protocol to reach a consensus on the value of the masked ballot $\hat{v_1}, ..., \hat{v_n}$.

3. The result of the vote is $\sum_i \hat{v_i} = \sum_i v_i(mod\ n+1)$.

3.6.2 Handling Dishonest Ballot Talliers

A Quantum Honest Success Byzantine Agreement Protocol (QHBA) is used in their voting scheme to identify dishonest ballot talliers.

Definition 5 (Honest success Byzantine agreement protocol (HBA)) *An honest success Byzantine agreement protocol involves n agents. One of the agents is the sender S, and holds an input value $x_s \in D$, where D is a finite domain. A protocol achieves honest success Byzantine agreement if the protocol guarantees the following:*

1. If the sender is honest, then all honest agents agree on the same output value $y = x_s$.

2. *If the sender is dishonest, then either all honest receivers abort the protocol, or all honest receivers decide on the same output value $y \in D$.*

The protocol is p-resilient if the protocol works when less than a fraction of p receivers are dishonest.

The QHBA is $\frac{m-2}{m}$-resilient. m is the number of receivers, and is more efficient than a classical HBA protocol when there are many dishonest receivers [Sun+18].

Distribution of Correlated Lists

The first phase of the QHBA protocol is for correlated lists to be distributed among the agents using quantum secure direct communication.

Let the sender be $S = P_1$. Each agent $P_i \in \{P_{\frac{n}{2}+1}, ..., P_n\}$ is tasked with distributing a list of numbers L_k^i to agent $P_k \in \{P_1, ..., P_{\frac{n}{2}}\}$ such that:

1. $|L_k^i| = l \; \forall \; k \in \{1, ..., n/2\}$, where l is a multiple of 6.
2. $L_1^i \in \{0, 1, 2\}^l$. $\frac{l}{3}$ numbers on L_1^i are 0. $\frac{l}{3}$ are 1. $\frac{l}{3}$ are 2.
3. $L_k^i \in \{0, 1\}^l \; \forall \; k \in \{2, ..., n/2\}$
4. $\forall \; j \in \{1, ..., l\}$, if $L_1^i[j] = 0$, then $L_2^i[j] = ... = L_{n/2}^i[j] = 0$
5. $\forall \; j \in \{1, ..., l\}$, if $L_1^i[j] = 1$, then $L_2^i[j] = ... = L_{n/2}^i[j] = 1$
6. $\forall \; j \in \{1, ..., l\}$, if $L_1^i[j] = 2$, then $\forall \; k \in \{2, ..., m\}$ the probability that $L_k^i[j] = 0$ and that $L_k^i[j] = 1$ are equal.

If the number of receivers that report non-compliant lists from a distributor passes a threshold, then that distributor is classified as dishonest.

Sequential Composition List Formation

Let the number of honest distributors be h. Then the agents perform the following sequential composition.

$$L_1 = L_1^{n/2+1}, ..., L_1^{n/2+h} \tag{3.43}$$

$$L_2 = L_2^{n/2+1}, ..., L_2^{n/2+h} \tag{3.44}$$

$$\vdots \tag{3.45}$$

$$L_{n/2} = L_{n/2}^{n/2+1}, ..., L_{n/2}^{n/2+h} \tag{3.46}$$

The constructed sequential composition of correlated lists is then $\mathbb{L}$.

$$\mathbb{L} = (L_1, ..., L_{n/2}) \tag{3.47}$$

Consensus

Assuming $h > \frac{n}{2}$, the following procedure can be used to reach a consensus.

First, the sender S sends a binary number $b_{1,k}$ and a list of numbers $ID_{1,k}$ to each receiver P_k. $ID_{1,k}$ should indicate all the positions on L_1 where $b_{1,k}$ appears to P_k. An honest sender will send the same list to all receivers.

Each P_k will compare the $b_{1,k}$ and $ID_{1,k}$ to their list L_k. If any honest P_k finds information that is not consistent, then P_k sends $\perp$ to the other receivers. Otherwise, P_k sends $b_{1,k}$ and $ID_{1,k}$ to the other receivers.

After all these messages have been received, each honest P_k checks the following:

1. If there were more than two agents who sent binary numbers and lists that were consistent with L_k but some had different binary numbers, P_k outputs $\perp$.

2. If more than two agents sent the same binary numbers and lists which were consistent with L_k, these agents are considered to be honest. P_k outputs the binary number provided by these honest agents.

3. If more than two agents sent the same binary numbers and lists which were consistent with L_k, any other agents are considered dishonest. If all of the dishonest agents sent $\perp$ to P_k, then P_k sets v_k to the binary value provided by the honest agents.

4. In all other cases, P_k outputs $\perp$.

Consensus is achieved if at least $\frac{n}{4}$ agents output the same bit value.

Suppose P_j were a dishonest receiver, and $j \geq 2$. P_j would want to send a binary number $b_{j,k}$ and list of numbers $ID_{j,k}$ which was consistent with L_k.On L_j, there are $\frac{l}{2}$ appearances of $b_{j,k}$. On L_1 there are only $\frac{l}{3}$ appearances of $b_{j,k}$. So, there are $\frac{l}{6}$ positions of discord x, where $L_1[x] = 2$. If P_j selects a discord position x then with probability $\frac{1}{2}$, $L_k[x] \neq b_{j,k}$. P_j has to avoid all discord positions in order to avoid being identified as dishonest. This has a $(\frac{2}{3})^{\frac{l}{3}}$ probability of success which is very small when l is large. This is rationale behind the checks made by P_k listed above.

3.7 Quantum Blockchain Using Entanglement in Time

Rajan, Del, and Matt Visser published a quantum system design that uses time entanglement to replace the data structure component of blockchain technology [RV19]. Their approach uses the nonseparability of entangled photons to simulate the links between blocks of data. The approach addresses the issue of blockchain scalability using quantum effects.

Multipartite states like the GHZ entangled state are used to create a chained data structure. In the most trivial example of the approach, the contents of a block might be represented by a pair of bits $r_1 r_2$. These contents are encoded into a temporal Bell state i.e.

$$|\beta_{r_1 r_2}>^{0,\tau} = \frac{1}{\sqrt{2}}(|0^0>|r_2^\tau> + (-1)^{r_1}|1^0>|\bar{r_2}^\tau>) \tag{3.48}$$

As records are created, they are encoded as blocks into temporal Bell states. These photons are created and absorbed at their respective times.

$$|\beta_{00}>^{0,\tau}, |\beta_{10}>^{\tau,2\tau}, |\beta_{11}>^{2\tau,3\tau}, etc. \tag{3.49}$$

The bit strings of the Bell states are then effectively "chained" together using entanglement in time. This is accomplished using a fusion process: Bell states are recursively projected into a growing temporal GHZ state. This can be accomplished using an entangled photon-pair production source, a delay line and a Polarizing Beam Splitter (PBS). For example, two Bell states could be fused into the four photon GHZ state:

$$\begin{aligned} & |\psi+>_{a,b}^{0,0} \otimes |\psi+>_{a,b}^{\tau,\tau} \\ \xrightarrow{\text{delay}} & |\psi+>_{a,b}^{0,\tau} \otimes |\psi+>_{a,b}^{\tau,2\tau} = \frac{1}{2}(|h_a^0 v_b^\tau> + |v_a^0 h_b^\tau>) \otimes (|h_a^\tau v_b^{2\tau}> + |v_a^\tau h_b^{2\tau}>) \\ \xrightarrow{\text{PBS}} & \frac{1}{2}(|h_a^0 v_b^\tau v_a^\tau h_b^{2\tau}> + |v_a^0 h_b^\tau h_a^\tau v_b^{2\tau}>) = |GHZ>^{0,\tau,\tau,2\tau} \end{aligned} \tag{3.50}$$

The four photons propagate in their own spatial modes and exist at different times, but are time entangled. The state of the blockchain at a given time $t = n\tau$ is:

$$\begin{aligned} & |GHZ_{r_1 r_2 ... r_{2n}}>^{0,\tau,\tau,2\tau,2\tau,...,(n-1)\tau,(n-1)\tau,n\tau} \\ = & \frac{1}{\sqrt{2}}(|0^0 r_2^\tau r_3^\tau ... r_{2n}^{n\tau}> + (-1)^{r_1} |1^0 \bar{r_2}^\tau \bar{r_3}^\tau ... \bar{r_{2n}}^{n\tau}>) \end{aligned} \tag{3.51}$$

This state contains the classical information $r_1 r_2 ... r_{2n}$. This information can be decoded without measuring the full photon statistics or detecting the photons [Meg+13b]. The scalability issue is addressed since "any number of photons can be generated with the same setup, solving the scalability problem caused by the previous need for extra resources. Consequently, entangled photon states of larger numbers than before are practically realizable" [Meg+13a].

3.8 Discussion

Ever since Wiesner's proposal of public-key quantum money in 1960 [250], quantum cryptography has been an active area of research. However, the topic of quantum blockchain is still relatively new. This is clear from a simple search for papers using the keywords "Quantum" and "Blockchain". There has been a steep, almost exponential, increase in publications over the last three years. There are still many open questions in the area. Ongoing research has identified and introduced new unanswered questions.

We are now beginning to see blockchain technology being adopted and trusted for critical government processes. For example, a prominent blockchain company ConsenSys Systems [Con19] has partnered with an initiative created by His Highness Sheikh Mohammad bin Rashid Al Maktoum, Vice President and Prime Minister of the UAE and Ruler of Dubai to use blockchain widely in Dubai [Con]. They released a whitepaper at the World Government Summit of 2017 entitled "Building the Hyperconnected Future on Blockchains" [New17]. Some companies that believe in the fundamental potential of decentralized governance like ConsenSys have endeavored to bring blockchain technology to areas of society that could be improved in some way by decentralization, with some success. ConsenSys has supported projects in decentralized journalism [Ile18], law [WR], digital asset economy [OM17], supply chains [Tre19] and more.

The longevity of technology that will impact our most important societal structures is worth questioning. There are critical issues with the scaling properties and efficiency of these blockchain technologies which require solutions if any significant distributed ledgers are going to be sustainably implemented. The scaling properties of the immutable distributed data structures used in blockchain networks have been shown to cause demands on memory that are hard to justify. Blockchains that are based on proof-of-work consensus schemes like Bitcoin also encourage massively wasteful resource consumption. Competition in Bitcoin's computationally-intensive scheme coupled with the limitations of the blockchain

data structure implementation by Bitcoin also causes issues with throughput of the system as a practical trading platform. The number of transactions that can be processed by Bitcoin is less than seven per second. This is far from the reported 47,000 per second achieved by VISA [VJR18].

These issues have motivated some pushback against the spread of blockchain. China is seeking to stop Bitcoin mining in the country, for example [Goh19]. From a business perspective, blockchain technology is not expected to be viable for full adoption and practical use by mainstream banks for around another ten years [Sch16]. Even so, banks are beginning to implement prototypes and blockchain applications of limited scale now. An IBM survey of 200 global banks [Mac17] showed that 65% of these banks intended to roll out blockchain-based products between 2016 and 2019.

The majority of blockchain applications that are being developed do not have solutions to the scalability and efficiency issues of their underlying cybersecurity schemes. They are also not prepared to face the challenges of attackers equipped with the quantum computers we expect to see developed within the next ten to twenty years. Companies are laying the groundwork now for technology that will become fundamentally tied to our most important societal structures, and this technology must be poised for viability in the quantum age. This is what has motivated efforts by companies like NXM Labs to introduce autonomous security protocols which can adapt and be securely updated to accommodate new challenges in the future [Inc19]. This is also what has motivated the Quantum Resistant Ledger project [MW].

In this review, we have focused on work that attempts to harness quantum computing to improve blockchain technology. These efforts are currently theoretical frameworks, but future quantum computing infrastructure may enable their realization. Their attempts to address the security and efficiency of blockchain cryptocurrencies [Wie83; BB87; Lut+09; Zha19], security primitives [Unr16b; ARU14; Feh18; Unr16a], smart contracts [Col19], consensus algorithms [Sun+18], and data structures [RV19] will inform and direct the future

implementations of quantum blockchain technologies.

Quantum money was arguably one of the first ideas that kickstarted the entire field of quantum cryptography in 1960 [Wie83]. The inherent security of information stored in quantum states using conjugate coding [Wie83] brings clear benefit to cryptocurrencies. Wiesner's basic scheme inspired the notable work of Bennett and Brassard's BB84 quantum key distribution protocol [BB87], among many other foundational works in the field. However, the conjugate coding scheme is not perfect for every use. Hence, improvements have been made such as the idea of collision-resistant quantum money introduced by Lutomirski, Andrew, et al [Lut+09] and Zhandry's Quantum Lightning framework [Zha19]. Despite its long history, quantum money still comes with some unanswered questions. Zhandry proved that any non-collapsing hash function can be used to construct Quantum Lightning. However, there are currently no known hash functions that are proven to be non-collapsing [Zha19]. It is an open question whether suitable hash functions could be constructed from better-known assumptions, such as the hardness of lattice problems.

Secure communication primitives have been relevant for work in quantum blockchain technology research. We have summarized the key points of foundational work on binding quantum commitment schemes [Unr16b; ARU14; Feh18; Unr16a]. Binding commitments underlie collision-resistant quantum money, Quantum Lightning. In turn, these are the primitives used by Coledangelo's hybrid quantum blockchain design [Col19]. Coledangelo's is one of the first hybrid quantum/classical blockchain designs, and notably addresses the issue of decohering quantum money by introducing a novel method for a blockchain participant to prove that they once owned a quantum coin. The design also includes a concept of arbitrary smart contracts much like Ethereum's. Open questions that remain for hybrid blockchain designers include the problem of ensuring the trustworthiness of arbitrary smart contract code and the hardness of the classical blockchain security elements against quantum attacks, among others.

The consensus algorithm presented by Sun, Xin, et al. [Sun+18] is a simple approach to

consensus which is adapted for use in a quantum blockchain. Their work demonstrates that consensus can be elegantly simple. Comparing the scaling characteristics of their scheme to those of other consensus algorithms may be useful for future works. Rajan, Del, and Matt Visser's blockchain data structure using entanglement in time [RV19] is an interesting, new perspective on quantum blockchain. Using a partially quantum mechanical data structure for blockchain may enable hybrid blockchain technologies to take advantage of effects such as entanglement swapping using photons [Meg+13a], and many violations of local realism. Whether it will become practical from an engineering or economic point of view to harness these effects on a large scale is yet to be determined.

The goal of this review was to provide a summary of current quantum blockchain research that can help to guide future work. There is huge potential for combining quantum resources with blockchain technology for applications in a variety of sectors including finance, healthcare, manufacturing and other areas where data security in a distributed network is of importance. We hope that this review will provide a resource to researchers from these different fields and enable further research and development.

Chapter Four

Arbitrary Quantum Code Execution

Streaming Web Code Execution and Verification on Quantum Annealers.

4.1 Chapter Overview

We suggest an approach to delegating the verification of arbitrary code to classical and quantum resources within a network. We look at the possibilities provided by WebAssembly (WASM) since it is an interoperable compilation target for other languages, meaning it can act as a "gateway" to executing code in a variety of languages. We present a methodology based on quantum annealing. Executing a simulation of a WASM function on a quantum annealer enables probabilistic edge-case detection which can be used to verify the correctness of the function. The method may benefit from a speedup due to quantum tunneling in the annealer. This suggests a possible efficiency improvement over the analogous classical sampling techniques being adopted by web software companies. Our approach however does not guarantee such a speedup. It is our hope that the methodology will enable further research to identify if and where quantum annealing may provide an edge over analogous classical methods.

4.2 Introduction

A fundamental barrier to using quantum technology in applied, commercial settings is the impracticality of interfacing with today's quantum backends which are complex to maintain and relatively unreliable in this early stage of their development. Quantum computers can now be accessed through the cloud programmatically by anyone. However, challenges with using the platform in any production environment include long wait times for experiments to be queued for the execution on a quantum device, high levels of noise and decoherence, as well as the fact that none of these computers can yet perform a computation that isn't simulatable using a high-end classical computer. On the other hand, D-Wave has made notable progress in building quantum annealing devices. While these are not general-purpose quantum computing devices, they are a much more scalable technology. The largest annealer

at the moment is D-Wave's 2000Q annealer with 2048 qubits [DWac]. D-Wave has also announced a new annealer with a 5000 qubit topology [DWaa]. These "qubits" are limited to a 2-local Ising model Hamiltonian which can be used to find a set of Boolean values that minimize a quadratic pseudo-Boolean function. In April 2019, a mapping of arbitrary Verilog code to a 2-local Ising model Hamiltonian form equivalent was published by Scott Pakin of Los Alamos National Laboratory [Pak19]. While Verilog is a domain-specific programming language used only in low-level hardware description projects, the published approach raises the question of what classical software could be useful to execute on a quantum annealer.

We argue that a specific domain of continuous test automation tasks that is immediately relevant to the verification of the correctness and security of large-scale commercial software development projects could benefit from improvements by delegating a particular class of tests to execution on a quantum annealer, using our method. We introduce a compilation technology that enables the execution of algorithms written in practical programming languages including Rust, C/C++, PHP, Python, Ruby, TypeScript and JavaScript on D-Wave's quantum annealer systems to this end. A logical next step for future work would be to provide a language-agnostic test automation framework for the validation of web-based software.

The contribution of this work is to automate the parallelization of arbitrary WASM code and the collapse of all of the dependencies in the stack machine-based assembly language to a set of simple feed-forward data dependency trees. This is done in order to identify the largest possible set of quadratic pseudo-Boolean function minimization problems that can be used to represent the segments of the original WASM code which can be made equivalent to Boolean expressions of combinational logic. This results in a set of QUBOs which can be simulated by a quantum annealer and ensures that the amount of the original WASM code that is made simulatable is maximized and can be source-mapped to the original code. The resulting simulations are combinational in nature and so are reversible. We explain how the reversibility of these simulations can be used to simplify security of correctness related

edge-cases detection, which could be automated in a web code testing framework.

4.3 Continuous Testing

Some understanding of continuous testing methodologies and quantum annealers will be required as we intend to bridge the gap between these two areas with our work. Continuous testing is the process of validating software before it is released through the execution of automated tests that are a part of the software delivery pipeline. The output of continuous testing is not simply a set of pass or fail data points, but an evaluation of the business risks associated with a software release. Continuous testing is a dev-ops practice which is typical in agile teams since agile is fundamentally about enabling teams to continuously deliver software rather than have large, spaced deadlines. 97% of organizations have adopted agile and 71% practice dev-ops according to Sauce Labs [PP18]. The demand for continuous testing automation tools has grown with this trend.

The main challenges that face companies who use current continuous test automation tools are the following according to Wolfgang Platz [PP18]: time/resources, complexity and results.

To define the time and resources required for continuous test automation, one must consider the time required for developers to write and maintain test scripts as well as the time and resources required to actually execute the test suite.

In production software, the complexity of test automation does not simply rely on the complexity of the algorithms under test. The predictable statefulness of test data, orchestration of numerous third-party technologies, and cost-effectiveness of running all of the necessary processes must be considered. It is a typical practice for companies to maintain an entire separately running instance of each of their software products for testing and validation purposes only, which can easily double the daily cost of dev-ops related spending. In the continuous delivery pipeline, the candidate code for release will be loaded into this

test environment to be automatically run against a test suite. Only if the test suite passes will the release be successful, and typically escalated for human approval.

Results of continuous testing processes are intended to enable informed decisions about the business risk of approving a release, and provide information that developers can use to diagnose and quickly solve technical problems. The diagnosis information relies on the tests that are included in the automated suite.

Types of tests that are used widely include: end-to-end tests, integration tests and unit tests. Each of these types of tests address a particular part of the software stack. End to end tests and integration tests involve testing that the pieces of a software solution developed individually work in a production environment when they are deployed together. Quantum annealers cannot help us with this unless they were both able to simulate the full complexity of the combined software system (which is beyond the reach of today's annealers) and if they were able to understand the diverse set of languages and environments that are used throughout a modern software product.

Unit tests are written by the developers of the functional code in an application to validate the behaviour of modular elements of the system. These require the developers to setup the relevant state of the application, provide input data to the module under test, execute the module under test, and write pass / fail conditions for the test. This is where developers must anticipate edge cases and manually specify the set of input data to provide to the module under test. Ideally, a module has constraints on its inputs that confine the space of possible inputs to reasonable values but this is most often not the case, especially in truly functional programming scenarios. It would also be ideal for all possible inputs to tested. This is never the case since that would require a large commitment of time from the developer writing each test and would require hugely inconvenient computational resources during the continuous testing step of the delivery pipeline.

There is a constant tug-of-war between the coverage of test cases and the practicality of implementing large numbers of tests. At the end of the day, the goal is to meaningfully

diagnose the problems with a release. This is often done efficiently by limiting the tests that are implemented to be the most meaningful tests. A meaningful test catches an edge condition that could plausibly occur, or validates the core functionality of a software module with a few instances of data; enough data points to be convincing.

Some companies have begun taking a sampling approach to testing and monitoring. One such company is Honeycomb. Honeycomb's founder describes the need for qualitative sampling techniques in an interview [YSS19]:

> "I need a tool that will work with that uncertainty and work with that flexibility rather than hemming me into the questions that I thought to ask ahead of time... Logs are no longer human scale, they're machine scale, and as a result, we can start to do things like sample intelligently and capture just enough to gain a sketch of what's happening in our system in real-time. " - Christine Yen (co-founder and CEO Honeycomb)

To address this tug-of-war, a few companies have also begun looking at introducing machine learning. Ubisoft is one of the companies that has introduced AI solutions to this problem [Led19]. Clever-commit is an AI system that analyzes incoming code and infers whether it is likely to cause software bugs by comparing it to a database of past commits that either passed or failed continuous testing. Clever-commit is accompanied by a handful of other similar AI-based solutions. Clever-commit is notable due to having been adopted by high-profile customers like Mozilla. Mozilla uses Clever-commit to ensure the quality of Firefox. The success of this type of project demonstrates that an informed suggestion about what code might cause problems can be more valuable to companies than a finite set of tests written by a developer who hopes to anticipate the possible problems with their own code. The introduction of an unbiased, automated approver that uses a more general evaluation technique can lead more directly and efficiently to a meaningful diagnosis.

4.4 Simulating Classical Software Using Quantum Annealers

Efforts have been made to generalize the classical logic systems that can be optimized on D-Wave's annealers. Most programming tools deal with very low-level logic systems. For example, macro assemblers called QMASM [Pak16] and qbsolv [DWa20] have syntax similar to assembly languages and essentially just directly specify the linear and quadratic coefficients of problems intended for D-Wave. ThreeQ.jl enables the construction of QUBOs within the Julia programming language [OV16]. In April 2019, Scott Pakin tackled a formidable task and introduced a method of compiling arbitrary Verilog code through a number of steps to QMASM that can be executed on a D-Wave system [Pak19]. This is the state-of-the-art in quantum annealer based simulation of classical programs written in traditional programming languages.

D-Wave also provides their own Python library for programming their annealers [DWad]. This library has recently introduced some higher-level methods that provide developers the ability to easily compose Hamiltonians that correspond to simulations of slightly more complex digital constructs. For example, library methods exist for creating simulations of combinational half and full adders.

4.5 WebAssembly

WebAssembly [Haa+17] is a low-level language that addresses the safety, speed and portability of code on the Web. The language is inherently safe and fast to execute. It is also language, hardware and platform-independent. the language has an efficient representation which is compact and easy to decode, validate and compile. It is also streamable and parallelizable.

The appeal of WebAssembly is largely that it has been designed so that it can be executed

directly in the browser. So, it is enabling more efficient and portable execution of code targetted at the web. This inherently platform-agnostic language is capable of enabling safer and more efficient execution of code in progressive web-based mobile applications on Android and iOS phones, web apps in any evergreen browser, Electron-based desktop applications for Windows, Mac, Linux operating systems, and more. Languages that can compile to WASM include Rust, C/C++, PHP, Python, Ruby, TypeScript and JavaScript, thanks to projects like Wasmer [Inc20].

The computational model of WASM is a stack machine. WASM code consists of lists of instructions that are executed in order. These instructions manipulate values on an implicit operand stack. Simple instructions perform basic operations on data found on the stack. Another type of instruction is control instructions. Control instructions introduces structure to a script using constructs like blocks, loops, and conditionals. Branches can only target such constructs.

WebAssembly would be an appealing language to compile to QMASM due to its compatibility with the technology and environments that are typically found to be used in projects that make use of deployment pipelines.

4.6 Approach

While WASM is more hardware-agnostic than other assembly languages like x86 and we don't have to deal with assumptions in the language about execution hardware, compiling arbitrary WASM code in its entirety to QMASM is still not a reasonable endeavor. Instead, we classify types of code blocks that can be simulated either in one shot, or as a part of a multi-step validation process. A fundamental difficulty in translating WASM to QMASM is the sequential nature of WASM code. We are limited to 2048 qubits for code and data, so this defines the boundary between what WASM modules will be simulatable and what modules need to be broken up further before being simulated.

Cell	Logic	Quadratic pseudo-Boolean function representation
NOT	$Y = \neg A$	$\mathbb{H}_{\neg}(\sigma_\gamma, \sigma_A) = \sigma_A\sigma_\gamma$
AND	$Y = A \wedge B$	$\mathbb{H}_{\wedge}(\sigma_\gamma, \sigma_A, \sigma_B) = -\frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B + \sigma_\gamma + \frac{1}{2}\sigma_A\sigma_B - \sigma_A\sigma_\gamma - \sigma_B\sigma_\gamma$
OR	$Y = A \vee B$	$\mathbb{H}_{\vee}(\sigma_\gamma, \sigma_A, \sigma_B) = -\frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B - \sigma_\gamma + \frac{1}{2}\sigma_A\sigma_B - \sigma_A\sigma_\gamma - \sigma_B\sigma_\gamma$
NAND	$Y = A \uparrow B$	$\mathbb{H}_{\uparrow}(\sigma_\gamma, \sigma_A, \sigma_B) = -\frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B - \sigma_\gamma + \frac{1}{2}\sigma_A\sigma_B + \sigma_A\sigma_\gamma + \sigma_B\sigma_\gamma$
NOR	$Y = A \downarrow B$	$\mathbb{H}_{\downarrow}(\sigma_\gamma, \sigma_A, \sigma_B) = \frac{1}{2}\sigma_A + \frac{1}{2}\sigma_B + \sigma_\gamma + \frac{1}{2}\sigma_A\sigma_B + \sigma_A\sigma_\gamma + \sigma_B\sigma_\gamma$
XOR	$Y = A \oplus B$	$\mathbb{H}_{\oplus}(\sigma_\gamma, \sigma_A, \sigma_B, \sigma_a) = \frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B - \frac{1}{2}\sigma_\gamma + \sigma_a - \frac{1}{2}\sigma_A\sigma_B - \frac{1}{2}\sigma_A\sigma_\gamma + \sigma_A\sigma_a + \frac{1}{2}\sigma_B\sigma_\gamma - \sigma_B\sigma_a - \sigma_\gamma\sigma_a$
XNOR	$Y = A \Leftrightarrow B$	$\mathbb{H}_{\Leftrightarrow}(\sigma_\gamma, \sigma_A, \sigma_B, \sigma_a) = \frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B + \frac{1}{2}\sigma_\gamma + \sigma_a - \frac{1}{2}\sigma_A\sigma_B + \frac{1}{2}\sigma_A\sigma_\gamma + \sigma_A\sigma_a - \frac{1}{2}\sigma_B\sigma_\gamma - \sigma_B\sigma_a + \sigma_\gamma\sigma_a$
2:1 MUX	$Y = (S \wedge B) \vee (\neg S \wedge A)$	$H_{MUX}(\sigma_\gamma, \sigma_S, \sigma_A, \sigma_B, \sigma_a) = \frac{1}{2}\sigma_S + \frac{1}{4}\sigma_A - \frac{1}{4}\sigma_B + \frac{1}{2}\sigma_\gamma + \sigma_a + \frac{1}{4}\sigma_S\sigma_A - \frac{1}{4}\sigma_S\sigma_B + \frac{1}{2}\sigma_S\sigma_\gamma + \sigma_S\sigma_a + \frac{1}{2}\sigma_A\sigma_B - \frac{1}{2}\sigma_A\sigma_\gamma + \frac{1}{2}\sigma_A\sigma_a - \sigma_B\sigma_\gamma - \frac{1}{2}\sigma_B\sigma_a + \sigma_\gamma\sigma_a$

Table 4.1 QMASM library combinational quadratic pseudo-Boolean functions

QMASM includes macros for many combinational digital primitives, some important examples are provided here.

We can enjoy the flexibility of not having our WASM executed on predefined combinational and sequential hardware. However, this flexibility is bounded by qubit limitation.

4.7 WebAssembly Code Analysis

Scott Pakin described a method of converting circuit netlists given in EDIF format [Sta01] to QMASM in his 2019 paper [Pak19]. The fundamental challenge in compiling WASM to QMASM may seem to be first expressing the WASM code in a format similar to a circuit's netlist. However, both EDIF and WASM's readable text file format .wat use the same fundamental S-expression [Riv94] tree-like data structure for their document formats. Like in EDIF, a WASM module in text format is represented by a single S-expression [Doc].

This similarity between EDIF and WASM will enable us to design a transformation of WASM to a netlist-inspired format that can be compiled to QMASM. The discrepancies between a netlist and a WASM S-expression come from the difference between WASM's sequential stack machine model and a netlist's combinational nature.

The ability to map data dependencies of a circuit netlist onto the Ising model does not address the full gambit of dependencies that we will find in a sequential language like WASM. To compile WASM, we have to consider each of the following dependencies:

- Data dependencies
- Name dependencies
- Control dependencies

The two types of name dependencies are anti-dependencies and output dependencies. Antidependence describes when a value is written to a memory location by an instruction that follows a previous instruction that used a value read from the same memory location, creating a WAR hazard. This is a non-issue in all properly implemented execution pipelines. However, since we are contriving something entirely different from a traditional processor execution pipeline, we will be careful to consider every possible issue.

The second type of name dependence is output dependence, which describes when two instructions write to the same memory location and create a WAW hazard.

Control dependence describes the case when an instruction's execution is dependent on the evaluation of a branch condition elsewhere in the code.

Sequential code can theoretically be simulated using QMASM by trading the time dimension for additional spatial dimensions, simulating every variable's value at each time-step individually. However, this is very costly in terms of qubits. Rather than directly simulating each variable's value for each time step, we endeavour to collapse all of the dependencies in a WASM program into a single data dependency tree that is expressible through the configuration of qubits in the Ising model.

4.8 Data Dependence

RAW data dependencies are expressed in EDIF by the sharing of a netlist node by two cells' input or output variables. Implementing a data dependency on a quantum annealer can be done by ensuring that the two variables' qubits' states are equal. These variables are the output from one process to its dependent process, and the dependent process' corresponding input variable. By solving for $N = 2$, we can show that it is straight forward to constrain two qubits' states to be equal in the trivial case where this is the only constraint. Setting $J_{i,j} = -1$, $h_i = h_j = 0$ we get the following evaluation.

$$\mathbf{H}(\sigma_i, \sigma_j) = h_i\sigma_i + h_j\sigma_j + J_{i,j}\sigma_i\sigma_j = -\sigma_i\sigma_j \tag{4.1}$$

As Scott Pakin pointed out, it is also easy to express the connection of multiple nodes using a Hamiltonian like the following, which connects one qubit σ_a to four others.

$$\mathbf{H}(\sigma_a, \sigma_b, \sigma_c, \sigma_d, \sigma_e) = -\sigma_a\sigma_b - \sigma_a\sigma_c - \sigma_a\sigma_d - \sigma_a\sigma_e \tag{4.2}$$

In WASM, data dependencies will need to be traced through transactions with the stack and with linear memory; the stateful elements of a WASM program. All simple WASM instructions pop arguments from the operand stack and push results back to it. Other instructions are needed to communicate with linear memory. For example, **get_local** loads information from a memory location in linear memory into the stack.

An example of the simple WASM function below shows how a function interacts with the stack and linear memory to perform a mathematical operation: the calculation of acceleration from initial velocity, final velocity and the elapsed time.

```
(func $accel (param $vi i32) (param $vf i32) (param $t i32) (result i32)
  (i32.div_u
    (i32.sub
      (get_local $vf)
      (get_local $vi)
    )
    (get_local $t)
  )
)
```

In JavaScript, this function might have been:

```
function accel(vi, vf, t){
    return (vi - vf)/t;
}
```

See how in the WASM instance of this function the data dependency of the i8.div_u

instruction on the output of the i8.sub instruction is made explicit by the nesting of instructions. In this sense, WASM seems to be an ideal intermediate language to use when attempting to lower JavaScript or other high level languages to something that can be encoded into a QUBO.

If we trace the path of each piece of data used by this function, we can see interactions with linear memory and the stack. Since it is not feasible to use our qubit resources to implement a stack or linear memory, we can simply collapse the data dependencies that exist through the stack and linear memory into dependencies directly between the instructions. We can then see that it is easy to categorize the inputs to functions as those variables that are loaded from linear memory, and the data dependencies between instructions as those variables that are passed through the stack between instructions.

When we compile such a function to QMASM, each instruction will become a combinational element of the simulated system. It will be a necessary step to implement each integer and floating-point WASM instruction as a QMASM macro, for example. Each simulated function will be a circuit that contains internal data dependencies between the macros it uses to define its combinational logic. These internal dependencies will be directly mapped from WASM's stack dependencies. The interfaces between simulated circuits will be directly mapped from WASM's linear memory dependencies. These are easy to identify while parsing structured WASM code since stack dependencies are expressed by nesting, and memory dependencies are expressed by the use of methods like **get_local**.

4.9 Name Dependence

Name dependencies are expressed in WASM by interactions with linear memory. Hazard-free name dependencies are relatively trivial to convert to data dependencies since a name dependency is simply a dependency of a block of code on the location of input data in linear memory. When two blocks of code interact through linear memory, one is essentially passing

data to the other. This is slightly more complex to parse than data passed over the stack since the relationship is not manifested in the nested structure of the code. However, name dependencies can be removed by replacing transactions over linear memory with transactions over the stack. In practice, this means moving blocks of code that interact over linear memory closer together so that they can directly call each other in a nested fashion. This might result in the duplication of code that might otherwise have appeared once in the source. This is indirectly a manifestation of the fundamental trade-off relationship between the size of the code and the complexity of the control structure.

4.10 Control Dependence

Converting control dependencies to data dependencies is not a new problem by any stretch. In fact, publications from as far back as 1983 [All+83] attempt to address this transformation. The fundamental approach presented in [All+83] is to replace all flow control instructions in a program with variable-dependent conditionals. This work addresses this transformation for each of the flow control instructions available in Fortran. The approach can be adapted for application to the flow control instructions of WASM. In WASM, the flow control instructions that are variable-based conditionals are **if** and **br_if**. The other flow control instructions are **nop, unreachable, block, loop, br, br_table, return, call** and **call_indirect**. We will consider **br, br_table, return, call_indirect** and **call** to be branching instructions. As per [All+83], we can replace all branches in a WASM program by branch relocation and branch removal. The former involves moving each branch that is nested in loops to the same nesting level as its branch target. The later involves replacing branches that are on the same level directly with equivalent variable-based conditionals.

This transformation can be algorithmically applied to a program by following these steps:

1. Parse the WASM code

2. Normalize all loops

3. Perform basic block analysis

4. Convert branches to variable-based conditionals

5. Optimize the resulting conditionals where possible

6. Collapse the resulting data dependencies where possible

7. Generate the parallel code

This is an implementation of an algorithm analogous to PFC [All+82], but specifically designed to parallelize sequential WASM code. This technology will not only be useful for executing WASM on quantum processors, but for compiling WASM optimally for execution on other parallel computing systems, like GPUs since it allows all the possible branches that can be taken through a program to be explicitly enumerated in variables. There will also be implications for hybrid quantum / classical computation, as the technology will make the cross-compilation of different segments of a WASM program for different processors. Portions of many programs will benefit from being delegated to a quantum processor.

An efficient WASM parser has been written in Rust and was made open source by software developer Yury Delendik at Mozilla in November 2018 [Del]. Rust is appealing to use for our work due to its safe, efficient execution and low-level nature. When working with Rust, we can be certain that we do not mishandle or ignore possible paths. We can also take advantage of Rust features such as its awareness of stack and heap (linear memory) space, and variable ownership to catch WAW, RAW or WAR hazards during our transformation of WASM code. In the transformation of large WASM programs, these factors will become very important. So, we build on the open-sourced work of Mozilla in our implementation.

WASM currently has an open threading proposal, which would be the first instance of any parallelization to come to the language. The possibility of implementing a more

fundamental OpenMP-compatible [Bal15] parallelization algorithm for WASM has also been discussed [Kir], but categorized as a feature of the language that will not be included in the first release of any accepted threading proposal, since the scope of threading itself is already large. Hence, we believe that our work is in the same spirit as the future direction of WASM. We are also certain that we are the first to implement a compiler like ours for WASM.

4.11 WebAssembly Parallelization

The first step to implementing the parallelization algorithm is to create a workflow for creating WASM files that our program will be able to accept. In this experiment, we write '.wat' files, which are WebAssembly text files with the nested format described. These will need to be compiled to true binary '.wasm' files. We use 'wat2wasm', which is a part of the WebAssembly Binary Toolkit [Fou19] to do this compilation. To ensure that the parallelization algorithm implementation is capable of processing a sufficiently complex WASM program, we use Conway's Game of Life in WASM [Log]. In a future work it would be interesting to use a larger application, like the National Cancer Institute's open-sourced WebAssembly based cancer genome viewer which was published in 2018 [FM18] as a testbench.

The individual instructions of a WASM binary file will need to be read into the parallelizing compiler. The first task will be be to identify functions. Since the goal is not for entire WASM programs to simulated on a quantum annealer, it is logical that the largest possible block type to be simulated will be functions. Starting with functions will also simplify the amount of source mapping that will be necessary to communicate to users what segments of their web code are being simulated or delegated.

The Rust parser provided by Yury Delendik of Mozilla provides implementations of a validating parser and operator parser. The validating parser is responsible for parsing the instructions that define the structure and flow of WASM code. The operator parser is

responsible for parsing those instructions which you would find in lines of code that are meant to be executed during runtime.

So, we may use the validating parser to ensure that our WASM code is valid, and to deduce where in the code we may find function bodies. We begin by simply sequentially reading each instruction of the WASM binary code until a function body delimiter is found. At this point, the parsing of the function body is delegated to the operator parser.

The operator parser allows us to handle each individual instruction's bytecode uniquely. In our parallelizing compiler, we categorize all instructions found within a function into these types:

- Function calls
- Control flow instructions
- Data flow instructions
- Data creation instructions
- Data mutation instructions

Each function is initially transformed into a data structure defined as a *Node*. A *Node* will include all of the information required to delegate its code to a processor that does not necessarily have any of the context that would be provided by executing the rest of the code. This means that it will need to have knowledge of its inputs and outputs (data couplings), its internally instantiated data (constants), calls to other functions, references to other blocks of code, its full list of instructions, and the same data for each child *Node* that represents a function or code block referenced in its instructions.

To expand this data structure beneath each function in a moderately complex WASM program would bloat the program's footprint significantly and not scale. Therefore, we provide an interface for selectively expanding the *Node* data structure tree by choosing functions that are delegation candidates.

The program identifies the combinational elements within each *Node*, which will be divided by control flow instructions like conditionals and branching instructions. These sets of combinational elements will each be transformed into boolean optimization problem expressions that could be expressed as compositions of QMASM macros.

4.12 Compiling Combinational Elements to QMASM

A few existing tools can be used for the compilation of our combinational elements to QMASM. The first is an open-source Python package called PyQUBO [Com]. This package aims to enable the conversion of flexible mathematically expressed constraints to valid QUBOs or Ising models programmatically. The next tool is qb2qmasm [Pak]. qb2qmasm is an open-source tool from Los Alamos National Laboratory which accepts the numerical notation format that D-Wave uses to denote Hamiltonian parameters and outputs equivalent symbolic QMASM code.

These tools could be used as a part of a compilation toolchain with our WASM parallelizing compiler to complete the compilation of WASM code to QMASM code. In this case, our Rust program is responsible for parallelizing the WASM code, selecting delegation candidate functions, collapsing the dependencies in these functions into a single data dependency tree, collapsing all reachable code in a selected path from the function into a single sequential script, and compiling the combinational elements of this script into mathematical expressions. These expressions could be written to a file as our program's output. This file would next serve as the input to a program which uses PyQUBO to generate the input for the final tool in the chain, which is qb2qmasm. At the end of the chain, we have a QMASM macro.

4.13 Software Implementation

Here we provide an explicit walk-through of the important components of the compilation of a simple WASM module. A black background denotes actual readout from the program during an interactive session. The module chosen demonstrates the basic function of the compiler.

The short WASM module in Appendix B.1 provides functions that compute the dot products of 2- and 3-dimensional vectors. This module was written for the purposes of this demonstration in the S-expression format.

After compilation to a true WASM binary, the same script becomes that of appendix B.2. It is the task of our new transpiler to compile this binary to a set of constraints that PyQUBO can use to create a quadratic unconstrained binary optimization problem. Using the WebAssembly Foundation's toolkit for VSCode [TF20], we can view the binary in a somewhat readable format.

We can compile this example by invoking the compiler using the following terminal command:

```
cargo run --example parallelize ./tests/parallelization/math.wasm
```

The compiler's entry-point is a file called *main.rs*, which uses a simple interface to our new *mapper* module to compile a given WASM file.

```
let buf: Vec<u8> = mapper.read_wasm(&args[1]).unwrap();
let nodes = mapper.map(buf);
```

This *mapper* module contains data structures that represent the various transformations of WASM programs throughout parallelization, dependency tree collapse and compilation to

simulatable transfer functions for D-Wave.

The first task that the mapper performs is to scan the WASM code, and identify the type and relevance of each instruction. An example of verbose output from our Rust program identifying instructions that imply data dependencies (blue), function calls (purple), combinationally simulatable instructions (green) and non-critical code (white) is provided in appendix B.3.

The command-line program prompts the user is they wish to parallelize each encountered function. If a user does wish to parallelize a function, then each type of block including loops, conditionals, branches and calls under it will be found recursively, and normalized. An interactive parallelization session is shown in appendix B.4.

The simplest example of a flow control dependency normalization algorithm is that which handles any encountered $if/else/end$ control sequences. These types of dependencies are normalized in one pass of the WASM code. In the case of an $if/else/end$ sequence, the program will create a separate $Node$ for each conditionally executable block of instructions. The program will also convert the control dependency to a data dependency. This is done by first creating a $Spin$ that represents the truth of the if condition, then creating a flow control coupling of this $Spin$ to another $Spin$ within each of the conditionally executable blocks' $Nodes$. The flow control coupling made to the if block's $Node$ will a direct coupling, while the coupling made the the $else$ block will be an anti-chain (meaning it is inverted). This is achieved by the code in appendix B.5.

At the end of this process, each $Node$ will have the following data structures populated, capturing all of the context necessary to execute each individual combinational $Node$ in a meaningful way without having to execute the entire program at once. This is in keeping with the philosophy of WASM, which is a streamable language, meaning it can be executed as it is delivered in chunks of code. It is also helpful for the hybridization of code execution. Each encountered instruction that is compatible with PyQUBO, and D-Wave's systems is transformed into an $AbstractExpression$ which represents its abstract mathematical

function. The set of *AbstractExpressions* created during the processing of a block becomes the contents of the corresponding *operations HashMap*. The *operations HashMap* data structure is provided in appendix B.6.

The *lower* method may be called on any *Node* to compile its *operations* to a *Constraint*. The *Constraint* data structure is provided in appendix B.7. This involves converting each *AbstractExpression* to a set of corresponding *PhysicalExpressions*, which are Boolean algebraic operators that can be mapped directly to a graph that can be minor-embedded into the topology of a D-Wave annealer. This transformation is functionally equivalent to compiling a mathematical statement to a Boolean statement which is equivalent. Currently, only a few expression elements are supported, which match those currently supported by PyQUBO. The *AbstractExpression* and *PhysicalExpression* data structures are provided in appendices B.8 and B.9.

Before choosing to compile each *Node*, the user is presented with the populated *Node* tree that represents each function. A simple example of a populated node from our demo is presented in appendix B.10. When a *Node* is lowered to a *Constraint*, the resulting data structure's format is equivalent to that of PyQUBO's *Expression* data structure. The constraints for multiplication operations are constructed using a combination of the Baugh-Wooley algorithm for n-bit multiplication [BW73] and a row adder tree to achieve optimal time and space complexity [LV84].

For example, consider the set of *Constraint* expressions that would capture the behaviour of the eight-bit multiplier that was represented in WASM as the $i8.mul$ instruction. The formulas for each output bit in terms of the inputs are given in appendix B.11. The expressions are ordered as the most to least significant output bits' formulas. Of course, the expressions listed here only capture one *AbstractExpression*. A *Node* may contain any number of them, and the *Nodes* our example each contain 7 or 8.

These *Constraint* expressions are not yet true constraints. In order to actually constrain the expressions, we have to specify a range of outputs. This is where a software tester would

specify an output range that constitutes a particular type of business logic error. Once the constrained expressions are submitted to PyQUBO and then D-Wave, the tester will be able to see what inputs to the system correspond with the erroneous output range due to the invertibility of the quantum simulation.

We explicitly demonstrated the compilation of one particular QUBO that would be a part of simulating our dot product module. The Python script in appendix B.12 constrains the third bit of an 8-bit multiplication to be 1. The result of this PyQUBO script is a list of external field and inter-qubit coupling strengths that corresponds almost exactly to D-Wave's "Qubist" format. This result is provided in appendix B.13. It is an optional step to use qb2qmasm to transform this format to a valid QMASM module, which may be used and re-used in a modular way by QMASM developers in the future. In order to actually have the code executed on a D-Wave computer, the only remaining mandatory step is to use D-Wave's automated minor-embedding tool to have the qubits in our Hamiltonian mapped onto their 2048 Chimera topology. This automation service is provided as a part of their Ocean API.

4.14 Discussion

When would this technique be more efficient to use than a set of specific classical tests? It depends on the project. Like machine learning-based approaches, our approach is not right for every company. On the other hand, there are team profiles that are a good match for the technology. For example, in the medical, scientific and financial industries having very specific, understandable tests of every module is often non-negotiable. In this case, such a solution would not replace these types of tests, but inform them. It is actually in these types of companies that adding a machine learning step to a delivery pipeline is not a significant additional overhead, since pipelines already take a long time to execute. For example, the test automation step of Xanadu's pipeline for their Python library StrawberryFields typically

took 2.5 hours to execute at one point in time, and they only run conventional unit and integration tests.

Another type of team that could benefit from this type of solution is a team that does not want to invest a lot of developers' time manually writing tests and is not producing code that will directly impact the well being of humans or stock markets (the opposite end of the spectrum).

The same teams that benefit from AI solutions to code quality assurance would likely benefit from a quantum code quality auditing process. Quantum annealers are a scalable technology that are reaching commercial viability, and have been already purchased from D-Wave by a handful of companies [DWab].

Most users of this technology don't have their own devices, but access D-Wave's systems through the cloud. Several limitations still exist for users of quantum annealing technology, including the time spent waiting for access to centralized quantum annealer devices, the probabilistic nature of imperfect experiment outcomes and limitations on data size.

These limitations narrow the practical applications of current quantum annealing technology to a very small space. A continuous testing automation solution is arguably in this space because of how these limitations align with the problem being solved:

1. Continuous testing is already a lengthy process and quantum annealing can be run in parallel with conventional tests without adding additional run time.

2. Qualitative or suggestive data is valuable to software engineers that are trying to make sense of pieces of a large codebase.

3. If quantum annealing tests were employed to test individual functional modules of code like conventional unit tests are, the effects of the limitations on the magnitude of outputs from the annealer could be minimized by choosing appropriate software modules. In these cases sampling from a continuous range of inputs and outputs can

provide more value than implementing tests for preconceived edge cases and executing them sequentially.

Beyond fitting within these limitations, using a quantum annealer for testing automation of this kind would provide benefits that no other testing solutions can. For example, by using a quantum annealer to probabilistically sample from the space of input and output variables, we can be certain that the system will automatically find every edge case after being run enough times. This is a lot better than the certainty we have about a biased developer predicting his/her own code's edge cases. The nature of the sampling is also ideal for implementing a configurable risk sensitivity on code modules, simply by setting the number of runs that will be executed on the quantum annealer to simulate each module. As the risk sensitivity is reduced, the number of runs is reduced, and the edge cases that are least probable to occur (and therefore have the least associated risk) are the first to be excluded. Also, since the input and output spaces are simulated together, the system could be capable of identifying the specific problematic inputs and suggesting test cases or input validation rules to developers. These are possibilities that are not feasible with any current testing frameworks.

4.15 Conclusion

The sampling and machine learning-based validation techniques being adopted by web development companies suggest an opportunity for quantum optimization in the testing automation space. Quantum annealers like D-Wave's machines are ideal for this application due to their availability, scalability and ability to perform code simulation and edge case detection. To bridge the gap between web code and quantum annealers, we have presented a novel methodology and software package for executing simulations of WASM functions on quantum annealers.

The potential quantum speedup due to tunneling in the quantum annealer is not

guaranteed by our approach, and it is an open question whether arbitrary functions can be encoded in such a way that they will take advantage of such a quantum speedup. However, we hope that the methodology presented may enable more exhaustive research into the viability of quantum annealing as a computational platform.

The approach can be seen as a "hybridization" tool, which takes a purely classical program and guides a user through its transformation into a set of hierarchically ordered subroutines, some of which can be simulated on a quantum annealer. The complexity of transpiling a subroutine for execution on an annealer is not known prior to lowering the subroutine, first to a circuit-level description and then to a set of constraints. This uncertainty makes it difficult to know whether the approach will create an overall improvement in execution time in any particular case. It is also not known whether quantum annealing is a platform which is yet ready to offer a true quantum speedup in arbitrary applications. The benefits of the approach presented here are therefore highly speculative. However, we believe that the approach may make it easier for researchers to perform hybridization experiments, which may well lead to answers to questions of quantum advantage.

Chapter Five

Summary and Outlook

5.1 Summary of Results

This thesis has touched on a number of areas of importance with regard to the development of a quantum internet. While the topic of the thesis is theoretical quantum information science, the chapters have a practical focus which reflects the unique status of the field today. Due to an availability of remote quantum information processing systems that has never been enjoyed by researchers before this decade, we were able to experimentally demonstrate and validate controlled quantum teleportation [KB] and controlled dense coding [HLG01] protocols for the first time. Our results are a unique contribution to the field that provide insight both into these protocols themselves and into the viability of quantum control on a modern superconducting quantum computing platform. There is still a lot of room for these modern systems to improve. For each experiment on the superconducting quantum computer, we also ran equivalent simulations on a classical processor. In the case of each controlled communication protocol, the controller was significantly more effective in the classical simulation: 7% in the case of controlled teleportation, 10% in the case of controlled dense coding. This is due to the combination of quantum gate errors, qubit errors and decoherence effects that currently limit superconducting quantum computers from reaching their theoretical potential.

Once quantum computing technology is sufficiently reliable for communication protocols, quantum communication networks will become viable. Already, a number of theoretical quantum communication network designs have been proposed. An interesting new area at the interface of quantum communication with quantum cryptography is quantum blockchain. We provide the first in-depth academic review of this new field. Blockchain technologies are on track to become a viable option for use in mission-critical network applications at financial institutions within the next ten years. Within this same timeframe, quantum computers are expected to reach viability. The benefits of using quantum resources in blockchain network schemes may become available "just in time", and so the theoretical work of designing

quantum and hybrid quantum / classical blockchains is being done now. This has included a purely quantum blockchain [RV19], a full hybrid quantum / classical blockchain design [Col19], a quantum consensus algorithm [Sun+18], various quantum cryptocurrencies [Wie83; Lut+09; Zha19] and security primitives [Unr16b; ARU14; Feh18; Unr16a]. These proposals range from immediately useful to more speculative. Wiesner's quantum money for example is the basis for QKD, which is an information-theoretically secure communication protocol that has commercially available implementations. The security primitives proposed are cryptographic commitment schemes, which are frameworks for cryptographic processes. The frameworks themselves are valid, but their implementation will depend on the underlying use of classical hash functions with certain properties, and some of the schemes require properties that have never been found. The voting scheme, consensus algorithm and hybrid quantum / classical blockchain proposed will rely on the viability of quantum money and quantum security primitives. Finally, the fully quantum blockchain data structure proposed using entanglement in time is possibly the most immediately implementatable from an experimental perspective. However, it has serious flaws as a useful method of information storage, since it encodes its data in the states of photons which are not easy to store or maintain.

With the assumption that quantum networks will eventually penetrate the world's communcations systems, it is inevitable that communications systems will begin to make use of both the newer quantum resources and legacy classical resources that are available. One of the most general tasks that an actor may wish to achieve in a computer network is to execute an arbitrary function. It's clearly a very complex task to optimally transpile an arbitrary function in an arbitrary programming language to execute optimally in a network with both quantum and classical resources. Instead of trying to tackle this hard prolem, we proposed a solution to a particular scenario. We proposed and demonstrated a basic transpilation technology that allows a user to choose whether or not it is desirable to target each subset of a function to a quatum annealer rather than a classical computer. The

transpiler is reponsible for identifying blocks of code that could theoretically be executed on an annealer, and for lowering the selected blocks to a format that a D-Wave quantum annealer could understand. We chose WebAssembly (WASM) as the input language to use, since it is a popular transpilation target for other languages. We believe that we were the first to propose and demonstrate our solution to this particular problem.

5.2 Future Outlook

We have contibuted novel theoretical and experimental work to three different but related areas within quantum information processing technology. Each area is still relatively new. Our first contribution established that available quantum technology is sufficient to demonstrate, but not yet ideal for implementing, controlled quantum communication protocols. Our next two chapters are even more speculative. In chapter three we provided the first in-depth outline of quantum blockchain research, a new field in quantum cryptography which is yet to prove its worth. In chapter four we proposed a potential starting point for researchers who wish to tackle the hard problems of hybrid quantum / classical computing, but did not solve these problems ourselves. There is still much work to be done on a variety of fronts before we can achieve the design of a viable quantum / classical internet. Some open problems that remain include the design of hybrid quantum / classical blockchain schemes that do not rely on the security assumptions of underlying security primitives like quantum lightning, the reduction of trust assumptions in quantum and hybrid quantum / classical communication networks, hybrid multi-party function evaluation and anonymous quantum communication channels. These problems will need to be addressed in the coming years if quantum communication technology is to reach commercialization, adoption and maturity on a scale similar to that of the classical internet.

Bibliography

[All+82] Allen et al. *PFC: A Program to Convert Fortran to Parallel Form*. Mar. 1982. URL: https://hdl.handle.net/1911/101547.

[All+83] J. R. Allen et al. "Conversion of control dependence to data dependence". In: *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL 83* (1983). DOI: 10.1145/567067.567085.

[ARU14] Andris Ambainis, Ansis Rosmanis, and Dominique Unruh. "Quantum Attacks on Classical Proof Systems: The Hardness of Quantum Rewinding". In: *2014 IEEE 55th Annual Symposium on Foundations of Computer Science* (2014). DOI: 10.1109/focs.2014.57.

[Atz15] Marcella Atzori. "Blockchain Technology and Decentralized Governance: Is the State Still Necessary?" In: *SSRN Electronic Journal* (2015). DOI: 10.2139/ssrn.2709713.

[Bal15] Pavan Balaji. "OpenMP". In: *Programming Models for Parallel Computing* (2015). DOI: 10.7551/mitpress/9486.003.0014.

[Bar+95] Barenco et al. "Elementary gates for quantum computation". eng. In: *Physical review. A, Atomic, molecular, and optical physics* 52.5 (1995), pp. 3457–3467. ISSN: 1050-2947.

[BB20] Charles H. Bennett and Gilles Brassard. "Quantum cryptography: Public key distribution and coin tossing". In: *arXiv e-prints*, arXiv:2003.06557 (Mar. 2020), arXiv:2003.06557. arXiv: 2003.06557 [quant-ph].

[BB87] Charles Bennett and Gilles Brassard. "Quantum public key distribution reinvented". In: *ACM SIGACT News* 18 (July 1987), pp. 51–53. DOI: 10.1145/36068.36070.

[BČL19] Artur Barasiński, Antonín Černoch, and Karel Lemr. "Demonstration of Controlled Quantum Teleportation for Discrete Variables on Linear Optical Devices". In: *Phys. Rev. Lett.* 122 (17 Apr. 2019), p. 170501. DOI: 10.1103/PhysRevLett.122.170501. URL: https://link.aps.org/doi/10.1103/PhysRevLett.122.170501.

[Ber+08] Guido Bertoni et al. "On the Indifferentiability of the Sponge Construction". In: *Advances in Cryptology - EUROCRYPT 2008 Lecture Notes in Computer Science* (2008), pp. 181–197. DOI: 10.1007/978-3-540-78967-3_11.

[BM97] Eli Biham and Tal Mor. "Bounds on Information and the Security of Quantum Cryptography". In: *Physical Review Letters* 79.20 (1997), pp. 4034–4037. DOI: 10.1103/physrevlett.79.4034.

[BRP18] Marcello Benedetti, John Realpe-Gómez, and Alejandro Perdomo-Ortiz. "Quantum-assisted Helmholtz machines: A quantum–classical deep learning framework for industrial datasets in near-term devices". eng. In: *Quantum Science and Technology* 3.3 (2018), pp. 034007–. ISSN: 2058-9565.

[But13] Vitalik Buterin. *Ethereum whitepaper*. 2013. URL: https://whitepaper.io/document/5/ethereum-whitepaper.

[BW73] C.r. Baugh and B.a. Wooley. "A Twos Complement Parallel Array Multiplication Algorithm". In: *IEEE Transactions on Computers* C-22.12 (1973), pp. 1045–1047. DOI: 10.1109/t-c.1973.223648.

[CCD15] Cristian S. Calude, Elena Calude, and Michael J. Dinneen. "Guest Column: Adiabatic Quantum Computing Challenges". In: *ACM SIGACT News* 46.1 (2015), pp. 40–61. DOI: 10.1145/2744447.2744459.

[Cho08] Vicky Choi. "Minor-embedding in adiabatic quantum computation: I. The parameter setting problem". In: *Quantum Information Processing* 7.5 (2008), pp. 193–209. DOI: 10.1007/s11128-008-0082-9.

[Col19] Andrea Coladangelo. "Smart contracts meet quantum cryptography". In: *arXiv e-prints*, arXiv:1902.05214 (Feb. 2019), arXiv:1902.05214. arXiv: 1902.05214 [quant-ph].

[Com] Recruit Communications. *PyQUBO*. URL: https://github.com/recruit-communications/pyqubo.

[Con] ConsenSys. *Smart Dubai*. URL: https://consensys.net/blockchain-use-cases/government-and-the-public-sector/smart-dubai/.

[Con19] ConsenSys. *Get to Know the ConsenSys Mesh*. Aug. 2019. URL: https://media.consensys.net/get-to-know-the-47-projects-that-make-up-the-consensys-mesh-478b7d3028c1.

[Del] Yury Delendik. *yurydelendik/wasmparser.rs*. URL: https://github.com/yurydelendik/wasmparser.rs.

[Doc] MDN Web Docs. *Understanding WebAssembly text format*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format#S-expressions.

[DWaa] D-Wave. URL: https://www.dwavesys.com/p-ress-releases/d-wave-previ-ews-next-generation-quantum-computing-platform.

[DWab] D-Wave. *D-Wave Customers*. URL: https://www.dwavesys.com/our-company/customers.

[DWac] D-Wave. *The D-Wave 2000Q^{TM} System*. URL: https://www.dwavesys.com/d-wave-two-system.

[DWad] D-Wave. *Wave Ocean Software Documentation*. URL: http://docs.ocean.dwavesys.com/.

[DWa20] D-Wave. *dwavesystems/qbsolv*. Feb. 2020. URL: https://github.com/dwavesystems/qbsolv.

[Edi17] AccessScience Editors. *Reaching the 50-qubit milestone in quantum computing*. 2020/5/11/ 2017. URL: https://www.accessscience.com/content/reaching-the-50-qubit-milestone-in-quantum-computing/BR1120171.

[Edi19] OSTechNix Editor. *Blockchain 2.0 - What Is Ethereum [Part 9]*. May 2019. URL: https://www.ostechnix.com/blockchain-2-0-what-is-ethereum.

[Fed08] Serguei Fedortchenko. "A quantum teleportation experiment for undergraduate students". In: (20160708).

[Feh10] Serge Fehr. "Quantum Cryptography". eng. In: *Foundations of Physics* 40.5 (2010), pp. 494–531. ISSN: 0015-9018.

[Feh18] Serge Fehr. "Classical Proofs for the Quantum Collapsing Property of Classical Hash Functions". In: *Theory of Cryptography Lecture Notes in Computer Science* (2018), pp. 315–338. DOI: 10.1007/978-3-030-03810-6_12.

[FM18] Richard Finney and Daoud Meerzaman. "Chromatic: WebAssembly-Based Cancer Genome Viewer". In: *Cancer Informatics* 17 (2018). DOI: 10.1177/1176935118771972.

[Fou19] WebAssembly Foundation. *WebAssembly Binary Toolkit*. May 2019. URL: github.com/WebAssembly/wabt.

[Fra20] Jake Frankenfield. *Proof of Stake (PoS)*. Jan. 2020. URL: https://www.investopedia.com/terms/p/proof-stake-pos.asp.

[Goh19] Brenda Goh. *China wants to ban bitcoin mining*. Apr. 2019. URL: http://www.reuters.com/article/us-china-cryptocurrency/china-wants-to-ban-Bitcoin-mining-idUSKCN1RL0C4.

[GR23] Goren Gordon and Gustavo Rigolin. "Generalized Quantum State Sharing". In: 73.6 (20060323). ISSN: 10502947.

[Gro96] Lov K. Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC 96* (1996). DOI: 10.1145/237814.237866.

[H18] Fiona H. *What is the Hamiltonian?* Apr. 2018. URL: https://support.dwavesys.com/hc/en-us/articles/360003684614-What-is-the-Hamiltonian-.

[Haa+17] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017* (2017). DOI: 10.1145/3062341.3062363.

[HLG01] Jiu-Cang Hao, Chuan-Feng Li, and Guang-Can Guo. "Controlled dense coding using the Greenberger-Horne-Zeilinger state". In: *Physical Review A* 63.5 (Nov. 2001). DOI: 10.1103/physreva.63.054301.

[Ile18] Matthew Iles. *The Civil White Paper*. Oct. 2018. URL: https://blog.joincivil.com/the-civil-white-paper-3e6c6f72dd9e.

[Inc19] NXM Labs Inc. *NXM Labs Announces Breakthrough in Quantum-Safe Security for Existing Computers and IoT Devices*. Apr. 2019. URL: http://www.newswire.ca/news-releases/nxm-labs-announces-breakthrough-in-quantum-safe-security-for-existing-computers-and-iot-devices-890174168.html.

[Inc20] Wasmer Inc. *wasmerio/wasmer*. Apr. 2020. URL: https://github.com/wasmerio/wasmer.

[KB] A. Karlsson and M. Bourennane. "Quantum Teleportation using Three Particle Entanglement". In: *Technical Digest. 1998 EQEC. European Quantum Electronics Conference (Cat. No.98TH8326)* (). DOI: 10.1109/eqec.1998.714867.

[Kir] J. A. Kirkham. *OpenMP . Issue 97 . WebAssembly/threads*. URL: https://github.com/WebAssembly/threads/issues/97.

[Kit97] A Yu Kitaev. "Quantum computations: algorithms and error correction". eng. In: *Russian Mathematical Surveys* 52.6 (1997), pp. 1191–1249. ISSN: 0036-0279.

[LD08] Xihan Li and Fuguo Deng. "Controlled teleportation". In: *Frontiers of Computer Science in China* 2.2 (2008), pp. 147–160. DOI: 10.1007/s11704-008-0020-0.

[Led19] Sylvestre Ledru. *Making the Building of Firefox Faster for You with Clever-Commit from Ubisoft*. Feb. 2019. URL: https://blog.mozilla.org/futurereleases/2019/02/12/making-the-building-of-firefox-faster-for-you-with-clever-commit-from-ubisoft/.

[Log] Scott Logic. *Writing WebAssembly By Hand*. URL: https://blog.scottlogic.com/2018/04/26/webassembly-by-hand.html.

[LSH19] Junjie Liu, Dvira Segal, and Gabriel Hanna. "Hybrid quantum-classical simulation of quantum speed limits in open quantum systems". eng. In: *Journal of physics. A, Mathematical and theoretical* 52.21 (2019), pp. 215301–. ISSN: 1751-8121.

[LU05] Marco Lanzagorta and Jeffrey K. Uhlmann. "Hybrid quantum-classical computing with applications to computer graphics". In: *ACM SIGGRAPH 2005 Courses on - SIGGRAPH 05* (2005). DOI: 10.1145/1198555.1198723.

[Lut+09] Andrew Lutomirski et al. "Breaking and making quantum money: toward a new quantum cryptographic protocol". In: *arXiv e-prints*, arXiv:0912.3825 (Dec. 2009), arXiv:0912.3825. arXiv: 0912.3825 [quant-ph].

[LV84] Wing K. Luk and Jean Vuillemin. "Recursive implementation of optimal time VLSi integer multipliers". In: 1984.

[Mac17] Tanaya Macheel. *Banks Will Start Actually Using Blockchain Next Year: IBM Report*. Jan. 2017. URL: https://www.americanbanker.com/news/banks-will-start-actually-using-blockchain-next-year-ibm-report.

[McG] Catherine C. McGeoch. *Adiabatic quantum computation and quantum annealing theory and practice*. Synthesis lectures on quantum computing, 8. Morgan and Claypool. ISBN: 1-68173-212-2.

[Meg+13a] E. Megidish et al. "Entanglement Swapping between Photons that have Never Coexisted". In: *Physical Review Letters* 110.21 (2013). DOI: 10.1103/physrevlett.110.210403.

[Meg+13b] E. Megidish et al. "Quantum tomography of inductively created multiphoton states". In: *Cleo: 2013* (2013). DOI: 10.1364/cleo_qels.2013.qf2b.6.

[Mer07] N. David Mermin. *Quantum Computer Science: An Introduction*. Cambridge University Press, 2007. DOI: 10.1017/CBO9780511813870.

[MW] Jack Matier and Pete Waterland. *Quantum Resistant Ledger (QRL)*. URL: https://raw.githubusercontent.com/theQRL/Whitepaper/master/QRL_whitepaper.pdf.

[Nak] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. URL: https://bitcoin.org/bitcoin.pdf.

[Neu+17] Florian Neukart et al. "Traffic Flow Optimization Using a Quantum Annealer". In: *Frontiers in ICT* 4 (2017). DOI: 10.3389/fict.2017.00029.

[New17] ETH News. *ConsenSys Releases Whitepaper At Dubai's World Government Summit*. Feb. 2017. URL: www.ethnews.com/consensys-releases-whitepaper-at-dubais-world-government-summit.

[Nof+17] Michael Nofer et al. "Blockchain". In: *Business & Information Systems Engineering* 59.3 (2017), pp. 183–187. DOI: 10.1007/s12599-017-0467-3.

[OM17] Michael Oved and Don Mosites. *Swap Protocol Whitepaper*. May 2017. URL: https://swap.tech/whitepaper/.

[OV16] Daniel Omalley and Velimir V. Vesselinov. "ToQ.jl: A high-level programming language for D-Wave machines based on Julia". In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)* (2016). DOI: 10.1109/hpec.2016.7761616.

[Pak] Scott Pakin. *qb2qmasm*. URL: https://github.com/lanl/qmasm/wiki/qb2qmasm.

[Pak16] Scott Pakin. "A quantum macro assembler". In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)* (2016). DOI: 10.1109/hpec.2016.7761637.

[Pak19] Scott Pakin. "Targeting Classical Code to a Quantum Annealer". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Apr. 2019). DOI: 10.1145/3297858.3304071.

[PP18] Wolfgang Platz and Wolfgang Platz. *3 biggest roadblocks to continuous testing*. July 2018. URL: https://www.infoworld.com/article/3294197/3-biggest-roadblocks-to-continuous-testing.html.

[Riv94] Ronald L. Rivest. *S-Expressions*. May 1994. URL: http://people.csail.mit.edu/rivest/Sexp.txt.

[RV19] Del Rajan and Matt Visser. "Quantum Blockchain Using Entanglement in Time". In: *Quantum Reports* 1.1 (2019), pp. 3–11. DOI: 10.3390/quantum1010002.

[Sch16] Paul Schaus. *Blockchain Projects Will Pay Off - 10 Years from Now*. Dec. 2016. URL: http://www.americanbanker.com/opinion/blockchain-projects-will-pay-off-10-years-from-now.

[SPS19] Rohit Singh, Harish Parthasarathy, and Jyotsna Singh. "Quantum image restoration based on Hudson–Parthasarathy Schrodinger equation". eng. In: *Quantum Information Processing* 18.11 (2019), pp. 1–29. ISSN: 1570-0755.

[Sta01] British Standards. "electronic design interchange format". In: (July 2001). DOI: 10.3403/bsen61690.

[Sun+18] Xin Sun et al. "A Simple Voting Protocol on Quantum Blockchain". In: *International Journal of Theoretical Physics* 58.1 (2018), pp. 275–281. DOI: 10.1007/s10773-018-3929-6.

[TF20] Dmitriy Tsvettsikh and WebAssembly Foundation. *wasmerio/vscode-wasm*. Feb. 2020. URL: https://github.com/wasmerio/vscode-wasm.

[Tre19] Treum. *Verified Organic and ConsenSys-backed Treum launch Ethereum blockchain solution to track and trace the first commercial hemp crop planted in Arizona*. June 2019. URL: https://itsupplychain.com/verified-organic-and-

consensys-backed-treum-launch-ethereum-blockchain-solution-to-track-and-trace-the-first-commercial-hemp-crop-planted-in-arizona/.

[UNM17] Hayato Ushijima-Mwesigwa, Christian F. A. Negre, and Susan M. Mniszewski. "Graph Partitioning using Quantum Annealing on the D-Wave System". In: *arXiv e-prints*, arXiv:1705.03082 (May 2017), arXiv:1705.03082. arXiv: 1705.03082 [quant-ph].

[Unr16a] Dominique Unruh. "Collapse-Binding Quantum Commitments Without Random Oracles". In: *Advances in Cryptology - ASIACRYPT 2016 Lecture Notes in Computer Science* (2016), pp. 166–195. DOI: 10.1007/978-3-662-53890-6_6.

[Unr16b] Dominique Unruh. "Computationally Binding Quantum Commitments". In: *Advances in Cryptology - EUROCRYPT 2016 Lecture Notes in Computer Science* (2016), pp. 497–527. DOI: 10.1007/978-3-662-49896-5_18.

[Vin+19] Walter Vinci et al. "A Path Towards Quantum Advantage in Training Deep Generative Models with Quantum Annealers". In: *arXiv e-prints*, arXiv:1912.02119 (Dec. 2019), arXiv:1912.02119. arXiv: 1912.02119 [quant-ph].

[VJR18] Dejan Vujicic, Dijana Jagodic, and Sinisa Randic. "Blockchain technology, bitcoin, and Ethereum: A brief overview". In: *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)* (2018). DOI: 10.1109/infoteh.2018.8345547.

[Wie83] Stephen Wiesner. "Conjugate coding". In: *ACM SIGACT News* 15.1 (1983), pp. 78–88. DOI: 10.1145/1008908.1008920.

[WR] Aaron Wright and David Roon. *A free legal repository*. URL: http://www.openlaw.io/.

[Yag+18] Dylan J. Yaga et al. *Blockchain Technology Overview*. Nov. 2018. URL: https://www.nist.gov/publications/blockchain-technology-overview.

[Yao+20] Yongxin Yao et al. "Gutzwiller Hybrid Quantum-Classical Computing Approach for Correlated Materials". In: *arXiv e-prints*, arXiv:2003.04211 (Mar. 2020), arXiv:2003.04211. arXiv: 2003.04211 [cond-mat.str-el].

[YSS19] Christine Yen, Adam Stacoviak, and Jerod Santo. *Observability Is for Your Unknown Unknowns with Christine Yen*. Aug. 2019. URL: https://changelog.com/podcast/356.

[Zeh70] H. D. Zeh. "On the interpretation of measurement in quantum theory". In: *Foundations of Physics* 1.1 (1970), pp. 69–76. DOI: 10.1007/bf00708656.

[Zha19] Mark Zhandry. "Quantum Lightning Never Strikes the Same State Twice". In: *Advances in Cryptology - EUROCRYPT 2019 Lecture Notes in Computer Science* (2019), pp. 408–438. DOI: 10.1007/978-3-030-17659-4_14.

APPENDIX

Appendix A

Controlled Communication Algorithms on IBM Quantum

A.1 Controlled Teleportation Protocol

```
from IBMQuantumExperience import IBMQuantumExperience
import random


API_TOKEN = '...'
FIDELITY_TEST_GATE = ["0","pi/2","-pi/2"] #params for the daggers of the
    teleported states


def
    controlled_teleport_sim(shots,bell_state,charlie,tele_theta,tele_phi,tele_lambda):
    '''
    Controlled teleportation test on the simulator.


    Open QASM:


        IBMQASM 2.0;
        include "qelib1.inc";
```

```
qreg q[5]; //define 5 quibit register, q[2]=A qubit, q[1]=B
    qubit,q[0]=C qubit
creg c[4]; //define 4 bit classical register

//creating a 3-qubit GHZ state

h q[0]; //perform hadamard on q[0]
cx q[0],q[1]; //CNOT on q[1] controlled by q[0]
cx q[1],q[2]; //CNOT on q[2] controlled by q[1]

//prepare state to be transfered. u3 =
    [[cos(theta/2),-exp(1i*lambda)*sin(theta/2)],[exp(1i*phi)*sin(theta/2),exp(1i*lamb
u3(tele_theta,tele_phi,tele_lambda) q[3];

//Alice performs Bell state measurement entangling x and A

cx q[3],q[2];
h q[3];

//measure the state of qubit C to obtain Rc

h q[1]; //perform hadamard on q[1]
measure q[1] -> c[0]; //measure q[1] into c[0], 0 corresponds to the
    outcome |+> and 1 corresponds to the |->

// Bobs U will be one of 4 gates depending on Rc and the bell
    measurement used
```

```
//1st bell state

if(c == 1) z q[0];
if(c == 0) I q[0];

//2nd bell state

if(c == 1) I q[0];
if(c == 0) z q[0];

//3rd bell state

if(c == 0) x q[0];
if(c == 1) z q[0];
if(c == 1) x q[0];

//4th bell state

if(c == 1) x q[0];
if(c == 0) z q[0];
if(c == 0) x q[0];

//measure Alices Bell state
measure q[2] -> c[2];
measure q[3] -> c[3];

//rotate the correct state to zero for fidelity test
```

```
    u3(FIDELITY_TEST_GATE[0],FIDELITY_TEST_GATE[1],FIDELITY_TEST_GATE[2])
        q[1];

    //measure the final transferred result
    measure q[0] -> c[1];

'''

api = IBMQuantumExperience.IBMQuantumExperience(API_TOKEN)
device = 'simulator'

#Experiment setup
qasm = "IBMQASM 2.0;\ninclude \"qelib1.inc\";\nqreg q[5];\ncreg c[4];\n"

#create GHZ state
qasm += "h q[0];\ncx q[0],q[1];\ncx q[1],q[2];\n"

#prepare state to be teleported u3 =
    [[cos(theta/2),-exp(1i*lambda)*sin(theta/2)],[exp(1i*phi)*sin(theta/2),exp(1i*lambda+
qasm += "u3({0},{1},{2}) q[3];\n".format(tele_theta,tele_phi,tele_lambda)

#Alice performs Bell state measurement entangling x and A
qasm += "cx q[3],q[2];\nh q[3];\n"

#Charlie measures the state of C
qasm += "barrier q[1];\n"
qasm += "h q[1];\nmeasure q[1] -> c[0];\n"
```

```
#Bob applies U
qasm += "barrier q[0],q[2],q[3];\n"

if(charlie == -1):
    charlie = int(random.getrandbits(1))

if(bell_state == 1):
    qasm += "if(c == {0}) z q[0];\n".format(int(not charlie))
elif(bell_state == 2):
    qasm += "if(c == {0}) z q[0];\n".format(int(not charlie))
elif(bell_state == 3):
    qasm += "if(c == {0}) x q[0];\nif(c == {1}) z q[0];\nif(c == {2}) x
        q[0];\n".format(int(charlie), int(not charlie), int(not charlie))
elif(bell_state == 4):
    qasm += "if(c == {0}) x q[0];\nif(c == {1}) z q[0];\nif(c == {2}) x
        q[0];\n".format(int(charlie), int(not charlie), int(not charlie))

#rotate correct state to zero for fidelity test
qasm += "u3({0},{1},{2})
    q[0];\n".format(tele_theta,FIDELITY_TEST_GATE[1],FIDELITY_TEST_GATE[2])

#measure the final result
qasm += "measure q[0] -> c[1];\n"

#measure the bell state
qasm += "measure q[2] -> c[2];\n"
qasm += "measure q[3] -> c[3];\n"
```

```
    exp = api.run_experiment(qasm, device, shots)

    return exp

def
    controlled_teleport_comp(shots,bell_state,charlie,tele_theta,tele_phi,tele_lambda):
    '''
    Controlled teleportation test on the quantum computer.
    The value for charlies measurement which we are interested in is passed as
        a parameter that
    is used to set the correct operations for Bob. This must also be taken
        into account in filtering
    for the relevant results later.

    Open QASM:

        IBMQASM 2.0;
        include "qelib1.inc";

        qreg q[5]; //define 5 quibit register, q[2]=A qubit, q[1]=B
            qubit,q[0]=C qubit
        creg c[4]; //define 4 bit classical register

        //creating a 3-qubit GHZ state

        h q[0]; //perform hadamard on q[0]
        cx q[0],q[1]; //CNOT on q[1] controlled by q[0]
        cx q[1],q[2]; //CNOT on q[2] controlled by q[1]
```

```
//prepare state to be transfered. u3 =
    [[cos(theta/2),-exp(1i*lambda)*sin(theta/2)],[exp(1i*phi)*sin(theta/2),exp(1i*lamb
u3(tele_theta,tele_phi,tele_lambda) q[3];

//Alice performs Bell state measurement entangling x and A

cx q[3],q[2];
h q[3];

//measure the state of qubit C to obtain Rc

h q[1]; //perform hadamard on q[1]
measure q[1] -> c[0]; //measure q[1] into c[0], 0 corresponds to the
    outcome |+> and 1 corresponds to the |->

// Bobs U will be one of 4 gates depending on Rc and the bell
    measurement used

//1st bell state

if(charlie == 1) z q[0];
if(charlie == 0) I q[0];

//2nd bell state

if(charlie == 1) I q[0];
if(charlie == 0) z q[0];
```

```
    //3rd bell state

    if(charlie == 0) x q[0];
    if(charlie == 1) z q[0];
    if(charlie == 1) x q[0];

    //4th bell state

    if(charlie == 1) x q[0];
    if(charlie == 0) z q[0];
    if(charlie == 0) x q[0];

    //measure Alices Bell state
    measure q[2] -> c[2];
    measure q[3] -> c[3];

    //rotate the correct state to zero for fidelity test
    u3(FIDELITY_TEST_GATE[0],FIDELITY_TEST_GATE[1],FIDELITY_TEST_GATE[2])
        q[1];

    //measure the final transferred result
    measure q[0] -> c[1];


'''

api = IBMQuantumExperience.IBMQuantumExperience(API_TOKEN)
```

```
device = 'real'


#Experiment setup
qasm = "IBMQASM 2.0;\ninclude \"qelib1.inc\";\nqreg q[5];\ncreg c[4];\n"


#create GHZ state
qasm += "h q[0];\ncx q[0],q[1];\ncx q[1],q[2];\n"


#prepare state to be teleported u3 =
    [[cos(theta/2),-exp(1i*lambda)*sin(theta/2)],[exp(1i*phi)*sin(theta/2),exp(1i*lambda+
qasm += "u3({0},{1},{2}) q[3];\n".format(tele_theta,tele_phi,tele_lambda)


#Alice performs Bell state measurement entangling x and A
qasm += "cx q[3],q[2];\nh q[3];\n"


#Charlie measures the state of C
qasm += "barrier q[1];\n"
qasm += "h q[1];\nmeasure q[1] -> c[0];\n"


if(charlie == -1):
    charlie = int(random.getrandbits(1))


#Bob applies U
qasm += "barrier q[0],q[2],q[3];\n"
if(bell_state == 1):
    if(not charlie):
        qasm += "z q[0];\n"
elif(bell_state == 2):
```

```
    if(charlie):
        qasm += "z q[0];\n"
elif(bell_state == 3):
    if(not charlie):
        qasm += "x q[0];\n"
    else:
        qasm += "z q[0];\nx q[0];\n"
elif(bell_state == 4):
    if(not charlie):
        qasm += "x q[0];\n"
    else:
        qasm += "z q[0];\nx q[0];\n"

#rotate correct state to zero for fidelity test
qasm += "u3({0},{1},{2})
    q[0];\n".format(tele_theta,FIDELITY_TEST_GATE[1],FIDELITY_TEST_GATE[2])

#measure the final result
qasm += "measure q[0] -> c[1];\n"

#measure the bell state
qasm += "measure q[2] -> c[2];\n"
qasm += "measure q[3] -> c[3];\n"

exp = api.run_experiment(qasm, device, shots)

return exp
```

A.2 Controlled Teleportation Experiment

```
from IBMQuantumExperience import IBMQuantumExperience
from protocols import *
import csv


API_TOKEN = '...'


def test_api_auth_token():
    '''
    Authentication with Quantum Experience Platform
    '''
    api = IBMQuantumExperience.IBMQuantumExperience(API_TOKEN)
    credential = api._check_credentials()


    return credential


def connect():
    '''
    Attempt to connect to the Quantum Experience Platform
    '''
    connection_success = test_api_auth_token()


    if(connection_success == True):
        print("API auth success.")
    else:
        print("API auth failure.")
        exit()
```

```
def run_cases(bell_states,charlie_expect,case_trials,device,prep_gate):
    total_set = 0
    total_clear = 0
    file_name = 'input_test_' + str(prep_gate[0].replace('/','over')) + '_result_'
        + device + '_' + str(case_trials) + 'x_' + 'c' + str(charlie_expect)

    f = open(file_name, 'w')
    if(device == 'comp'):
        csvFile = open('allCompData.csv','a')
    elif(device == 'sim'):
        csvFile = open('allSimData.csv','a')

    fieldnames = ["Zero","One","Alice's Relevance","Alice's
        Irrelevance","Charlie's Relevance","Charlie's Irrelevance","Bell
        State","Charlie Expect","Data State", "Executions", "Device"]
    writer = csv.DictWriter(csvFile, fieldnames=fieldnames)

    if(device == 'comp'):
        print("Controlled teleportation on the ibmqx2.")
        f.write("Controlled teleportation on the ibmqx2.\n")
    elif(device == 'sim'):
        print("Controlled teleportation on the simulator.")
        f.write("Controlled teleportation on the simulator.\n")

    for bell_state in bell_states:
        if(device == 'comp'): exp =
            controlled_teleport_comp(case_trials,bell_state,charlie_expect,prep_gate[0],prep_gate
            #teleport with Alice measuring ZW Bell state
```

```
if(device == 'sim'): exp =
    controlled_teleport_sim(case_trials,bell_state,charlie_expect,prep_gate[0],prep_gate[


p_set = 0
p_clear = 0
charlie_correct = 0
alice_correct = 0

if(bell_state == 1):
    bell_indicator = (0,1)
elif(bell_state == 2):
    bell_indicator = (0,0)
elif(bell_state == 3):
    bell_indicator = (1,0)
else:
    bell_indicator = (1,1)

print(exp)

print("Taking case where Alice measures Bell State {0} and Charlie
    measures {1}".format(bell_state,charlie_expect))
f.write("Taking case where Alice measures Bell State {0} and Charlie
    measures {1}\n".format(bell_state,charlie_expect))

if 'result' in exp:
    for i in range(len(exp['result']['measure']['labels'])):
        if((int(exp['result']['measure']['labels'][i][0]) ==
            charlie_expect) or (charlie_expect == -1)): #Charlies
```

```
        measurement is correct or he is not allowing this transmission
        charlie_correct += exp['result']['measure']['values'][i]
        if((int(exp['result']['measure']['labels'][i][2]) ==
            bell_indicator[0]) and
            (int(exp['result']['measure']['labels'][i][3]) ==
            bell_indicator[1])): #Alices measurement is correct!
            alice_correct += exp['result']['measure']['values'][i]
            if(int(exp['result']['measure']['labels'][i][1]) == 1): #and
                the result is a 1
                p_set += float(exp['result']['measure']['values'][i])
            elif(int(exp['result']['measure']['labels'][i][1]) == 0):
                #and the result is a 0
                p_clear += float(exp['result']['measure']['values'][i])

print("--------------------------")
print("TRANSFERRED STATE")
print("--------------------------")
print("With respect to full results")
print("--------------------------")
f.write("--------------------------\n")
f.write("TRANSFERRED STATE\n")
f.write("--------------------------\n")
f.write("With respect to full results\n")
f.write("--------------------------\n")

print("state   probability")
print("0       {0}".format(p_clear))
print("1       {0}".format(p_set))
```

```
f.write("state   probability\n")
f.write("0       {0}\n".format(p_clear))
f.write("1       {0}\n".format(p_set))

print("--------------------------")
print("In post selection subset")
print("--------------------------")
f.write("--------------------------\n")
f.write("In post selection subset\n")
f.write("--------------------------\n")

if((p_set+p_clear) != 0):
    p_set = p_set/(p_set+p_clear)
    p_clear = 1 - p_set
else:
    p_set = 0
    p_clear = 0

print("state   probability")
print("0       {0:.2f}".format(p_clear))
print("1       {0:.2f}".format(p_set))
f.write("state   probability\n")
f.write("0       {0:.2f}\n".format(p_clear))
f.write("1       {0:.2f}\n".format(p_set))

print("--------------------------")
print("Context relevance")
print("--------------------------")
```

```
        f.write("--------------------------\n")
        f.write("Context relevance\n")
        f.write("--------------------------\n")

        print("Alice's Bell Measurement")
        print("relevant: {0:.2f}".format(alice_correct))
        print("irrelevant: {0:.2f}".format(1 - alice_correct))
        f.write("Alice's Bell Measurement\n")
        f.write("relevant: {0:.2f}\n".format(alice_correct))
        f.write("irrelevant: {0:.2f}\n".format(1 - alice_correct))

        print("Charlie's Measurement")
        print("relevant: {0:.2f}".format(charlie_correct))
        print("irrelevant: {0:.2f}".format(1 - charlie_correct))
        f.write("Charlie's Measurement\n")
        f.write("relevant: {0:.2f}\n".format(charlie_correct))
        f.write("irrelevant: {0:.2f}\n".format(1 - charlie_correct))

    else:
        print('BAD API RESPONSE!')
        print(exp)
        p_set = 0
        p_clear = 0

    writer.writerow({
        "Zero": "{0:.2f}".format(p_clear),
        "One": "{0:.2f}".format(p_set),
        "Alice's Relevance": "{0:.2f}".format(alice_correct),
```

```
        "Alice's Irrelevance": "{0:.2f}".format(1 - alice_correct),
        "Charlie's Relevance": "{0:.2f}".format(charlie_correct),
        "Charlie's Irrelevance": "{0:.2f}".format(1 - charlie_correct),
        "Bell State": bell_state,
        "Charlie Expect": charlie_expect,
        "Data State": str(prep_gate[0].replace('/','over')),
        "Executions": case_trials,
        "Device": device
    })

    total_set += p_set
    total_clear += p_clear

if((total_set+total_clear) == 0):
    total_set = 0
    total_clear = 0
else:
    total_set = total_set/(total_set+total_clear)
    total_clear = 1 - total_set

print("--------------------------")
print("OVERALL RELEVANT RESULT")
print("--------------------------")
print("state   probability")
print("0       {0:.2f}".format(total_clear))
print("1       {0:.2f}".format(total_set))
f.write("--------------------------\n")
f.write("OVERALL RELEVANT RESULT\n")
```

```
    f.write("-------------------------\n")
    f.write("state   probability\n")
    f.write("0       {0:.2f}\n".format(total_clear))
    f.write("1       {0:.2f}\n".format(total_set))

    f.close()
    csvFile.close()

    return


connect() #connect to IBM Q

teleport_prep_gates = [["pi/6", "-pi/2", "pi/2"],["pi/5", "-pi/2",
    "pi/2"],["pi/4", "-pi/2", "pi/2"],["pi/3", "-pi/2", "pi/2"],["pi/2", "-pi/2",
    "pi/2"],["pi", "-pi/2", "pi/2"]] #parameters for the states to teleport
case_trial_array = [100] #run each case this many times to estimate resulting qubit
charlie_expect_array = [1] #expect charlie to measure this state for post selection
device_array = ['sim'] #device to run program on
bell_states = [3] #Alices bell states to test

for gate in teleport_prep_gates:
    for i in range(len(charlie_expect_array)):
        for j in range(len(case_trial_array)):
            for k in range(len(device_array)):

                charlie_expect = charlie_expect_array[i]
                case_trials = case_trial_array[j]
```

```
            device = device_array[k]

            run_cases(bell_states,charlie_expect,case_trials,device,gate)
```

A.3 Controlled Dense Coding Protocol

```
from IBMQuantumExperience import IBMQuantumExperience
import random

API_TOKEN = '...'

def dense_coding(device,shots,message):
    '''
    Dense coding test on the simulator.

    Open QASM:

        IBMQASM 2.0;
        include "qelib1.inc";

        qreg q[5]; //define 5 quibit register, q[2]=A qubit, q[1]=B qubit,q[0]=C
            qubit
        creg c[5]; //define 5 bit classical register

        //creating a 3-qubit GHZ state

        h q[2]; //perform hadamard on q[2]
```

```
cx q[2],q[1]; //CNOT on q[2] controlled by q[1]
cx q[1],q[0]; //CNOT on q[1] controlled by q[0]

//measure the state of qubit C to obtain Rc

h q[0]; //perform hadamard on q[1]
measure q[0] -> c[0]; //measure q[0] into c[0], 0 corresponds to the
    outcome |+> (allowed) and 1 corresponds to the |-> (disallowed)

//Alice performs a Pauli operator corresponding to message

I q[2]; // 00
x q[2]; // 10
y q[2]; // 11
z q[2]; // 01

barrier q[2];

// Bobx performs cx

cx q[2],q[1];

//Bob measures bits

measure q[1] -> c[1]; //parity bit determines whether operation is in
    {I,z} or {x,y}
```

```
    h q[2]; //other bit, 0 corresponds to the outcome |+> and 1 corresponds to
        the |->, giving the exact operation Alice performed
    measure q[2] -> c[2];

'''

api = IBMQuantumExperience(API_TOKEN)

#Experiment setup
qasm = "IBMQASM 2.0;\ninclude \"qelib1.inc\";\nqreg q[5];\ncreg c[5];\n"

#create GHZ state
qasm += "h q[2];\ncx q[2],q[1];\ncx q[1],q[0];\n"

#Cliff measures the state of qubit C to obtain Rc
qasm += "barrier q[0];\n"
qasm += "h q[0];\nmeasure q[0] -> c[0];\n"

#Alice encodes message
if(message == '10'):
    qasm += "x q[2];\n"
elif(message == '11'):
    qasm += "y q[2];\n"
elif(message == '01'):
    qasm += "z q[2];\n"

qasm += "barrier q[2];\n"
```

```
    #Bob performs cx

    qasm += "cx q[2],q[1];\n"

    #Bob measures bits

    #measure the parity bit
    qasm += "measure q[1] -> c[1];\n"

    #measure the final bit
    qasm += "h q[2];\n"
    qasm += "measure q[2] -> c[2];\n"

    exp = api.run_experiment(qasm, device, shots)

    return exp
```

A.4 Controlled Dense Coding Experiment

```
from IBMQuantumExperience import IBMQuantumExperience
from protocols import *
import csv

API_TOKEN = '...'

def test_api_auth_token():
    '''
```

```
    Authentication with Quantum Experience Platform
    '''
    api = IBMQuantumExperience(API_TOKEN)
    credential = api.check_credentials()

    return credential

def connect():
    '''
    Attempt to connect to the Quantum Experience Platform
    '''
    connection_success = test_api_auth_token()

    if(connection_success == True):
        print("API auth success.")
    else:
        print("API auth failure.")
        exit()

def run_cases(cliff_expect,case_trials,device,message,f,writer):
    total_correct = 0
    total_incorrect = 0

    if(device == 'ibmqx4'):
        print("Controlled dense coding on the ibmqx4.")
        f.write("Controlled dense coding on the ibmqx4.\n")
    elif(device == 'simulator'):
        print("Controlled dense coding on the simulator.")
```

```
    f.write("Controlled dense coding on the simulator.\n")

exp = dense_coding(device,case_trials,message)

print(exp)

p_correct = 0
p_incorrect = 0
cliff_correct = 0

print("Taking case where Alice sends {0} and cliff measures
    {1}".format(message,cliff_expect))
f.write("Taking case where Alice sends {0} and cliff measures
    {1}\n".format(message,cliff_expect))

if 'result' in exp:
    for i in range(len(exp['result']['measure']['labels'])):
        parity_bit = str(exp['result']['measure']['labels'][i][2])
        phase_bit = str(exp['result']['measure']['labels'][i][3])
        print("parity: {0}".format(parity_bit))
        print("phase: {0}".format(phase_bit))
        if((int(exp['result']['measure']['labels'][i][4]) ==
            (not(cliff_expect))) or (cliff_expect == -1)): #cliffs measurement
            is correct or he isn't allowing this transmission
            cliff_correct += exp['result']['measure']['values'][i]
            if((parity_bit == message[1]) and (phase_bit == message[0])): #and
                the result is correct
                p_correct += float(exp['result']['measure']['values'][i])
```

```
        else:
            p_incorrect += float(exp['result']['measure']['values'][i])

print("--------------------------")
print("TRANSFERRED MESSAGE")
print("--------------------------")
print("With respect to full results")
print("--------------------------")
f.write("--------------------------\n")
f.write("TRANSFERRED MESSAGE\n")
f.write("--------------------------\n")
f.write("With respect to full results\n")
f.write("--------------------------\n")

print("message   probability")
print("{0}       {1}".format(message,p_correct))
f.write("message   probability\n")
f.write("{0}       {1}\n".format(message,p_correct))

print("--------------------------")
print("In post selection subset")
print("--------------------------")
f.write("--------------------------\n")
f.write("In post selection subset\n")
f.write("--------------------------\n")

if((p_correct+p_incorrect) != 0):
    p_correct = p_correct/(p_correct+p_incorrect)
```

```
        p_incorrect = 1 - p_correct
    else:
        p_correct = 0
        p_incorrect = 0

    print("message   probability")
    print("{0}       {1:.2f}".format(message,p_correct))
    f.write("message   probability\n")
    f.write("{0}       {1:.2f}\n".format(message,p_correct))

    print("--------------------------")
    print("Context relevance")
    print("--------------------------")
    f.write("--------------------------\n")
    f.write("Context relevance\n")
    f.write("--------------------------\n")

    print("cliff's Measurement")
    print("relevant: {0:.2f}".format(cliff_correct))
    print("irrelevant: {0:.2f}".format(1 - cliff_correct))
    f.write("cliff's Measurement\n")
    f.write("relevant: {0:.2f}\n".format(cliff_correct))
    f.write("irrelevant: {0:.2f}\n".format(1 - cliff_correct))

else:
    print('BAD API RESPONSE!')
    print(exp)
    p_correct = 0
```

```
        p_incorrect = 0

    writer.writerow({
        "Success": "{0:.2f}".format(p_correct),
        "Cliff's Relevance": "{0:.2f}".format(cliff_correct),
        "Cliff's Irrelevance": "{0:.2f}".format(1 - cliff_correct),
        "Cliff Expect": cliff_expect,
        "Executions": case_trials,
        "Device": device
    })

    return

connect() #connect to IBM Q

messages = ['00','01','10','11'] #the messages to encode
case_trial_array = [1000] #run each case this many times to estimate result
device_array = ['ibmqx4'] #devices to run program on
cliff_expect_array = [1,-1] #expect cliff to measure this state for post selection

for k in range(len(device_array)):

    device = device_array[k]
    if(device == 'ibmqx4'):
        csvFile = open('allibmqx4Data.csv','a')
    elif(device == 'simulator'):
        csvFile = open('allSimData.csv','a')
```

```
    fieldnames = ["Success","Cliff's Relevance","Cliff's Irrelevance","Cliff
        Expect","Executions", "Device"]
    writer = csv.DictWriter(csvFile, fieldnames=fieldnames)
    writer.writeheader()

    for i in range(len(cliff_expect_array)):
        for j in range(len(case_trial_array)):
            for message in messages:

                cliff_expect = cliff_expect_array[i]
                case_trials = case_trial_array[j]

                file_name = 'coding_test_' + message + '_result_' + device + '_' +
                    str(case_trials) + 'x_' + 'c' + str(cliff_expect)
                f = open(file_name, 'w')

                run_cases(cliff_expect,case_trials,device,message,f,writer)

f.close()
csvFile.close()
```

Appendix B

WASM Transpiler

B.1 WAT Dot Product Module

```
(module
    (func $dot_three (param $vx i8) (param $vy i8) (param $vz i8) (param $wx i8)
        (param $wy i8) (param $wz i8) (result i8)
        (i8.add
            (i8.mul
                (get_local $vx)
                (get_local $wx)
            )
            (call $dot_two
                (get_local $vy)
                (get_local $vz)
                (get_local $wy)
                (get_local $wz)
            )
        )
    )
    (func $dot_two (param $vx i8) (param $vy i8) (param $wx i8) (param $wy i8)
        (result i8)
        (i8.add
```

```
            (i8.mul
                (get_local $vx)
                (get_local $wx)
            )
            (i8.mul
                (get_local $vy)
                (get_local $wy)
            )
        )
    )
    (export "dot_three" (func $dot_three))
    (export "dot_two" (func $dot_two))
)
```

B.2 WASM Dot Product Module

```
(module
    (type $t0 (func (param i8 i8 i8 i8 i8 i8) (result i8)))
    (type $t1 (func (param i8 i8 i8 i8) (result i8)))
    (func $dot_three (type $t0) (param $p0 i8) (param $p1 i8) (param $p2 i8)
        (param $p3 i8) (param $p4 i8) (param $p5 i8) (result i8)
    get_local $p0
    get_local $p3
    i8.mul
    get_local $p1
    get_local $p2
    get_local $p4
```

```
    get_local $p5
    call $dot_two
    i8.add)
    (func $dot_two (type $t1) (param $p0 i8) (param $p1 i8) (param $p2 i8) (param
        $p3 i8) (result i8)
    get_local $p0
    get_local $p2
    i8.mul
    get_local $p1
    get_local $p3
    i8.mul
    i8.add)
    (export "dot_three" (func $dot_three))
    (export "dot_two" (func $dot_two)))
)
```

B.3 Instruction Categorization

```
BeginWasm { version: 1 }
BeginSection { code: Type, range: Range { start: 10, end: 29 } }
TypeSectionEntry(FuncType { form: Func, params: [I32, I32, I32, I32, I32, I32], returns: [I32] })
TypeSectionEntry(FuncType { form: Func, params: [I32, I32, I32, I32], returns: [I32] })
EndSection
BeginSection { code: Function, range: Range { start: 31, end: 34 } }
EndSection
BeginSection { code: Export, range: Range { start: 36, end: 59 } }
ExportSectionEntry { field: "dot_three", kind: Function, index: 0 }
ExportSectionEntry { field: "dot_two", kind: Function, index: 1 }
EndSection
BeginSection { code: Code, range: Range { start: 61, end: 95 } }
BeginFunctionBody { range: Range { start: 63, end: 81 } }
1. Unreachable
2. GetLocal { local_index: 0 }
3. GetLocal { local_index: 3 }
4. I32Mul
5. GetLocal { local_index: 1 }
6. GetLocal { local_index: 2 }
7. GetLocal { local_index: 4 }
8. GetLocal { local_index: 5 }
9. Call { function_index: 1 }
10. I32Add
11. End
BeginFunctionBody { range: Range { start: 82, end: 95 } }
1. Unreachable
2. GetLocal { local_index: 0 }
3. GetLocal { local_index: 2 }
4. I32Mul
5. GetLocal { local_index: 1 }
6. GetLocal { local_index: 3 }
7. I32Mul
8. I32Add
9. End
EndSection
First pass found 2 functions:
[0, 1]
```

B.4 Interactive Function Parallelization

```
Parallelize function 1 (yes/no)?
y
Analyzing function 1...
Found 0 blocks in function 1
Found 0 calls to other functions from function 1
Parallelize function 0 (yes/no)?
y
Analyzing function 0...
Found 0 blocks in function 0
Found 1 calls to other functions from function 0
Registering call to function 1 from function 0
Found 0 blocks in function 1
Found 0 calls to other functions from function 1
```

B.5 *if/else/end* Control Sequences Handling

```
Operator::If { ty } => {
    stdout.set_color(ColorSpec::new().set_fg(Some(Color::Yellow)));
    print!("==== New If Condition: ");
    println!("{}. {:?}", i, op);

    // if conditions imply a single data dependency
    let mut conditional_node = Node::default();

    // create variable to represent the condition
    let outer_var_id = node.add_internal_variable(i, *ty);

    // create data coupling to simulate flow control
    let inner_var_id = conditional_node.add_input_variable(*ty);
    conditional_node.add_flow_control_coupling(outer_var_id, inner_var_id,
        true);
```

```
    // recursively process the conditional code block as a new node
    conditional_node = self.map_helper(reader, buf, resources, position, i,
        conditional_node);

    // register the conditional block
    let conditional_id = self.add_block(conditional_node.clone());
    node.add_block(i, conditional_id);

    // add a spin to each node
    node.add_operation(i, AbstractExpression::Spin{ id: outer_var_id });
    conditional_node.add_operation(i, AbstractExpression::Spin{ id:
        inner_var_id });

    stdout.set_color(ColorSpec::new().set_fg(Some(Color::Yellow)));
    print!("==== End of: ")
}
Operator::Else => {

    stdout.set_color(ColorSpec::new().set_fg(Some(Color::Yellow)));

    // else implies a single data anti-dependency
    // it needs to be constructed from within the if so we can have easy
        access to its coupling parameters
    // however, it will be lifted out during the collapse of its top-level
        parent function

    // we should have most recently registered a conditional node with only
        one flow control coupling
```

```
let couplings = node.get_flow_control_couplings();
let coupling_count = couplings.keys().len();

// we should have most recently registered a conditional node with only
    one input variable
let input_variables = node.get_input_variables();
let input_variable_count = input_variables.keys().len();

// if we aren't in a conditional already, don't process the else
if (coupling_count == 1 && input_variable_count == 1) {

    print!("==== New Else Clause: ");
    println!("{}. {:?}", i, op);

    // get coupling details from the if condition details
    let coupled_var_id = node.get_first_flow_control_coupling();
    let input_type = node.get_first_input_variable();

    let mut else_node = Node::default();

    // create data anti-chain coupling to simulate flow control
    let inner_var_id = else_node.add_input_variable(input_type);
    else_node.add_flow_control_coupling(coupled_var_id, inner_var_id,
        false);

    // recursively process the conditional code block as a new node
    else_node = self.map_helper(reader, buf, resources, position, i,
        else_node);
```

```
            // the else's end also terminates the if clause
            let if_end = else_node.get_end();
            node.set_end(if_end);

            // register the else block
            let else_id = self.add_block(else_node);
            node.add_block(i, else_id);

            stdout.set_color(ColorSpec::new().set_fg(Some(Color::Yellow)));
            print!("==== End of: ");
            println!("{}. {:?}", i, op);

            // finish processing the if node
            break;
        }
    }
```

B.6 Operations HashMap Data Structure

```
/// A node represents a segment of WASM code
/// These include functions and blocks at first,
/// then are transformed to combinational segments
/// of code after parallelization.
#[derive(Clone, Debug)]
pub struct Node {
```

```
id: usize, // each function and block has an id
instrs: Vec<u8>, // hex instructions of the node
branches: HashMap<usize, usize>, // internal locations and targets of branches
calls: HashMap<usize, usize>, // calls to other functions
start: usize, // where the node's insturctions start in the WASM source file
end: usize, // where the node's insturctions end in the WASM source file
children: HashMap<usize, Node>, // calls to other functions, or internal
    blocks of code
constants: HashMap<usize, Type>, // constants instantiated within the scope of
    the node
chains: HashMap<usize, Type>, // whether the spins at indeces i are coupled
    via chaining or anti-chaining
internal_variables: HashMap<usize, Type>, // internal variables that will be
    used to simulate flow control
input_variables: HashMap<usize, Type>, // all input variables including
    parameters, memory references, global references are given ids
output_variables: HashMap<usize, Type>, // all output varibles including
    writes to memory and returns
global_input_data_couplings: HashMap<usize, usize>, // map of global variable
    locations to the coupled node's input variable ids
global_output_data_couplings: HashMap<usize, usize>, // map of global variable
    locations to the coupled node's output variable ids
flow_control_couplings: HashMap<usize, usize>, // map of instruction locations
    to coupled flow control variable ids
input_data_couplings: HashMap<usize, usize>, // map of memory locations to the
    coupled node's input variable ids
output_data_couplings: HashMap<usize, usize>, // map of memory locations to
    the coupled node's output variable ids
```

```
    blocks: HashMap<usize, usize>, // internal blocks' locations mapped to their
        ids as maintained by the mapper
    operations: HashMap<usize, AbstractExpression> // simulatable operations
}
```

B.7 Constraint Data Structure

```
/// A Constraint represents a nestable constraint
/// expression.
#[derive(Clone, Debug)]
pub struct Constraint {
    id: usize, // maps each Constraint to its node
    expression: Option<PhysicalExpression> // low level boolean expressions
}
```

B.8 AbstractExpression Data Structures

```
/// The abstract operation enum represents logical operations
/// that can be compiled to simulatable transfer functions
/// for quantum annealers.
#[derive(Clone, Debug)]
pub enum AbstractExpression {
    Spin { id: usize },
    Num { val: usize },
```

```
    Add { ty: Type },
    Mul { ty: Type }
}
```

B.9 PhysicalExpression Data Structures

```
/// The physical expression enum represents the valid
/// operations and data types that can be understood by PyQUBO.
#[derive(Clone, Debug)]
pub enum PhysicalExpression {
    Not{ operand: Box<PhysicalExpression> },
    Add{ operand_one: Box<PhysicalExpression>, operand_two:
        Box<PhysicalExpression> },
    Mul{ operand_one: Box<PhysicalExpression>, operand_two:
        Box<PhysicalExpression> },
    Spin{ val: bool }, // 0 represents -1
    Num{ val: usize },
    Binary{ val: bool }
}
```

B.10 Simple Populated Node

```
0x1: Node {
    id: 0x1,
    instrs: [
        0x0,
        0x20,
        0x0,
        0x20,
        0x2,
        0x6c,
        0x20,
        0x1,
        0x20,
        0x3,
        0x6c,
        0x6a,
        0xb
    ],
    branches: {},
    calls: {},
    start: 0x52,
    end: 0x5f,
    children: {},
    constants: {},
    internal_variables: {},
    input_variables: {
        0x2: I32,
        0x1: I32,
        0x0: I32,
        0x3: I32
    },
    output_variables: {
        0x0: I32
    },
    global_input_data_couplings: {},
    global_output_data_couplings: {},
    flow_control_couplings: {},
    input_data_couplings: {},
    output_data_couplings: {},
    blocks: {},
    operations: {
        0x7: Mul {
            ty: I32
        },
        0x2: Spin {
            id: 0x0
        },
        0x8: Add {
            ty: I32
        },
        0x6: Spin {
            id: 0x3
        },
        0x4: Mul {
            ty: I32
        },
        0x3: Spin {
            id: 0x2
        },
        0x5: Spin {
            id: 0x1
        }
    }
},
```

B.11 i8.mul Constraints

```
(Spin("A5") * Spin("B5"))
```

```
(Not(Spin("A4") * Spin("B5")) + Not(Spin("A5") * Spin("B4")))
```

```
(Not(Spin("A3") * Spin("B5")) + Spin("A4") * Spin("B4") + Not(Spin("A5") *
    Spin("B3")))
```

```
(Not(Spin("A2") * Spin("B5")) + Spin("A3") * Spin("B4") + Spin("A4") * Spin("B3")
    + Not(Spin("A5") * Spin("B2")))
```

```
(Num(1) + Not(Spin("A1") * Spin("B5")) + Spin("A2") * Spin("B4") + Spin("A3") *
    Spin("B3") + Spin("A4") * Spin("B2") + Not(Spin("A5") * Spin("B1")))
```

```
(Not(Spin("A0") * Spin("B5")) + Spin("A1") * Spin("B4") + Spin("A2") * Spin("B3")
    + Spin("A3") * Spin("B2") + Spin("A4") * Spin("B1") + Not(Spin("A5") *
    Spin("B0")))
```

```
(Spin("A0") * Spin("B4") + Spin("A1") * Spin("B3") + Spin("A2") * Spin("B2") +
    Spin("A3") * Spin("B1") + Spin("A4") * Spin("B0"))
```

```
(Spin("A0") * Spin("B4") + Spin("A1") * Spin("B3") + Spin("A2") * Spin("B2") +
    Spin("A3") * Spin("B1") + Spin("A4") * Spin("B0"))
```

```
(Spin("A0") * Spin("B3") + Spin("A1") * Spin("B2") + Spin("A2") * Spin("B1") +
    Spin("A3") * Spin("B0"))
```

```
(Spin("A0") * Spin("B2") + Spin("A1") * Spin("B1") + Spin("A2") * Spin("B0"))

(Spin("A0") * Spin("B1") + Spin("A1") * Spin("B0"))

(Spin("A0") * Spin("B0"))
```

B.12 PyQUBO Constraining Third Bit of i8.mul Output

```
from pyqubo import Constraint, Spin

// Declare qubits
A0, B2, A1, B1, A2, B0 = Spin('A0'), Spin('B2'), Spin("A1"), Spin("B1"),
    Spin("A2"), Spin("B0")

// Optional strength of the constraint, or importance of the corresponding problem
M = 1.0

// Tell the compiler that (A0*B2+A1*B1+A2*B0-1)**2 is a constraint which should be
    zero when the solution is not broken
// This is equivalent to requiring that A0*B2+A1*B1+A2*B0 is one
constraint = Constraint((A0*B2+A1*B1+A2*B0-1)**2, 'error_condition')

// Define the Hamiltonian in terms of its objective function and constraint
exp = A0*B2+A1*B1+A2*B0 + M * constraint
model = exp.compile()
qubo = model.to_qubo()
```

```
print(qubo)
```

B.13 Ising Hamiltonian Parameters for Constrained i8.mul Problem

```
({('A2', 'A2'): -6.0
('A0*B2', 'A0*B2'): 27.0
('A0', 'A0*B2'): -10.0
('A1*B1', 'B0'): -16.0
('A1', 'B2'): 8.0,
('A0', 'B0'): 8.0
('B2', 'B2'): -6.0
('A1*B1', 'B1'): -10.0
('A2', 'A2*B0'): -10.0
('A0*B2', 'B1'): -16.0,
('A1', 'B1'): 5.0
('A2*B0', 'A2*B0'): 27.0
('A1', 'A2'): 8.0
('B1', 'B2'): 8.0
('A0*B2', 'A1*B1'): 32.0,
('A1*B1', 'A2'): -16.0
('A1', 'A2*B0'): -16.0
('A1', 'A1'): -6.0
('A0', 'A2'): 8.0
('A0', 'A0'): -6.0,
('A2*B0', 'B2'): -16.0
('A1', 'A1*B1'): -10.0
('A0', 'B1'): 8.0
('A1*B1', 'B2'): -16.0
('A2', 'B1'): 8.0,
('A0*B2', 'A2*B0'): 32.0
('A2*B0', 'B1'): -16.0
('B0', 'B2'): 8.0
('A2*B0', 'B0'): -10.0
('B1', 'B1'): -6.0,
('A0*B2', 'A1'): -16.0
('A0', 'A1'): 8.0
('A1*B1', 'A2*B0'): 32.0
('A0*B2', 'A2'): -16.0
('A0', 'A2*B0'): -16.0,
('A2', 'B0'): 5.0
('A0', 'B2'): 5.0
('A0*B2', 'B2'): -10.0
('B0', 'B1'): 8.0
('A1*B1', 'A1*B1'): 27.0,
('A0', 'A1*B1'): -16.0
('A2', 'B2'): 8.0
('A1', 'B0'): 8.0
('A0*B2', 'B0'): -16.0
('B0', 'B0'): -6.0}, 7.0)
```