Solving binary $\mathcal{MQ}$ with Grover's algorithm

Peter Schwabe and Bas Westerbaan *

Digital Security Group, Radboud University
peter@cryptojedi.org bas@westerbaan.name

Abstract. The problem of solving a system of quadratic equations in multiple variables—known as multivariate-quadratic or $\mathcal{MQ}$ problem—is the underlying hard problem of various cryptosystems. For efficiency reasons, a common instantiation is to consider quadratic equations over $\mathbb{F}_2$. The current state of the art in solving the $\mathcal{MQ}$ problem over $\mathbb{F}_2$ for sizes commonly used in cryptosystems is enumeration, which runs in time $\Theta(2^n)$ for a system of n variables. Grover's algorithm running on a large quantum computer is expected to reduce the time to $\Theta(2^{n/2})$. As a building block, Grover's algorithm requires an "oracle", which is used to evaluate the quadratic equations at a superposition of all possible inputs. In this paper, we describe two different quantum circuits that provide this oracle functionality. As a corollary, we show that even a relatively small quantum computer with as little as 92 logical qubits is sufficient to break $\mathcal{MQ}$ instances that have been proposed for 80-bit pre-quantum security.

Keywords: Grover's algorithm, multivariate quadratics, quantum resource estimates

1 Introduction

The effects of large quantum computers on the world of modern cryptography are often summarized roughly as follows: *"All factoring-based and discrete-log based cryptosystems are broken in polynomial time by Shor's algorithm [Sho94,Sho97]"* and *"symmetric crypto is affected by Grover's algorithm [Gro96], but we just have to double the key size"*. A more detailed look also reveals applications of Grover's algorithm in various asymmetric schemes (as in this paper); an even more detailed look considers the question what exactly "large quantum computer" means, i.e., how many logical qubits and how much time is required to implement Shor's and Grover's algorithm. In the following, when we say "time" we always refer to the cumulative number of gates that need to be executed. This is obviously very different from the number of gates that might be physically implemented. For

* This work has been supported by the European Commission through the ICT program under contract ICT-645622 (PQCRYPTO); by the European Research Council under grant 320571 (QCLS) and by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114. Permanent ID of this document: 40eb0e1841618b99ae343ffa073d6c1e. Date: 2017-11-30

example, implementing a loop of length 100 around a certain circuit increases the number of executed gates (i.e., the time) by a factor of 100, but does not increase the number of physical gates (except maybe for the loop counter).

Recently, multiple papers have taken this more detailed approach of analyzing the cost of quantum attacks against cryptographic primitives. For example, in [GLRS16], Grassl, Langenberg, Roetteler, and Steinwandt describe how to attack AES-128 with Grover's algorithm using a quantum computer with 2953 logical qubits in time about 2^{87}. We note that with the results of [GLRS16] it would also be possible to perform this computation on a quantum computer with only 984 qubits, however, then increasing time by a factor of 3. In [AMG+16], Amy, Di Matteo, Gheorghiu, Mosca, Parent and Schanck describe how to compute SHA-2 preimages with Grover's algorithm on a quantum computer with 2402 logical qubits in time about 2^{148} and how to compute SHA-3 preimages using 3200 qubits in time about 2^{154}. For Shor's algorithm the common estimate is that one needs approximately $2n$ qubits to factor an n-bit number[1]. Breaking RSA-1024 thus needs a quantum computer with at least 2048 logical qubits.

These results seem to suggest that quantum computers only affect cryptography once they can be scaled to at least about one thousand qubits. In this paper we show that *much smaller* quantum computers can be used to break cryptographic schemes. Ironically, the schemes we are targeting are "post-quantum" schemes, i.e., schemes that have been proposed to replace factoring-based systems like RSA and discrete-log based systems like DSA to resist attacks by quantum computers. Specifically, we describe how to use Grover's algorithm to solve multivariate systems of equations over $\mathbb{F}_2$. This problem is known as the $\mathcal{MQ}$ problem and it is in general NP-complete [GJ79]. It is the underlying hard problem of various signature schemes like HFEv$^-$ [PCG01,PCY+15] and (variants of) Unbalanced Oil-and-Vinegar (UOV) [KPG99,DS05], and the identification scheme proposed in [SSH11].

It is long known that Grover's algorithm provides a square-root speedup in enumeration attacks against this problem. What is new in this paper are two implementations together with a detailed analysis of the cost of this attack in terms of the number of required qubits and time (in the number of gates). These numbers for Grover's algorithm are largely determined by the number of qubits and time required in an oracle that evaluates the target function. In the case of $\mathcal{MQ}$, evaluating the target function means evaluating the system of quadratic equations at a superposition of all possible inputs. In this paper we describe two such oracles for systems of quadratic equations over $\mathbb{F}_2$. The first oracle is easy to describe and for $m-1$ quadratic equations in $n-1$ variables it only needs $m+n+2$ qubits and at most $2m(n^2+2n)+1$ gates executed. The second oracle is more sophisticated and requires only $3+n+\lceil \log_2 m \rceil$ qubits, but approximately double the number of gates executed of the first oracle.

As a consequence, we show that the "80-bit secure" parameters (80 equations in 84 variables) used, for example, in the identification scheme described

[1] The problem of factoring a number N is reduced to finding the order of an element x modulo N, which requires a bit more than $2\log_2 N$ qubits [NC10, §5.3.1].

in [SSH11] can be broken on a quantum computer with only 168 logical qubits in time about 2^{62} or on a quantum computer with only 94 logical qubits in time about 2^{63}.

Organization of this paper. Section 2 gives a very brief introduction to quantum computing to establish notation and to give the basic background required to follow the remainder of the paper. Section 3 collects the quantum gates we need in our oracles. Section 4 describes in detail our first Grover oracle for the $\mathcal{MQ}$ problem over binary fields with a careful analysis of the complexity. Section 5 continues with a description of the more complex second oracle which requires fewer qubits. Finally, in Section 6, we briefly sketch how to optimize for circuit depth instead of number of qubits. In Appendix A we provide quipper code to generate the oracles and Python code to generate the first oracle. We place this code into the public domain.

2 Preliminaries

In this section we will first give a concise definition of the problem we solve in this paper. Then we introduce the bare essentials of quantum computing to apply Grover's algorithm. For a proper introduction, see [NC10].

2.1 Problem definition

Problem 1. A system of quadratic equations over $\mathbb{F}_2$ is given by a "cube" $(\lambda_{ij}^{(k)})_{i,j,k}$ over $\mathbb{F}_2$ and a vector $(v_1, \ldots, v_m) \in \mathbb{F}_2^m$. The goal is to find $(x_1, \ldots, x_n) \in \mathbb{F}_2^n$ such that

$$\sum_{1 \leq i,j \leq n} \lambda_{ij}^{(1)} x_i x_j = v_1 \quad \ldots \quad \sum_{1 \leq i,j \leq n} \lambda_{ij}^{(m)} x_i x_j = v_m.$$

Note that the system also contains linear terms as $x_i^2 = x_i$.

For sizes of this problem commonly used in cryptography, the best classical algorithm known is (Gray-code) enumeration [BCC+14]. Specifically, [YCC04, Section 2.2] estimates that asymptotically faster algorithms take over only for systems with about $n = 200$ variables. On a quantum computer, however, one can use Grover's algorithm [Gro96,BHT98]. To apply Grover's algorithm, we need to provide a suitable *oracle*: a quantum circuit that checks whether a vector (x_i) is a solution for a given system $(\lambda_{ij}^{(k)}), (v_k)$. Every Boolean circuit can be translated into an equivalent quantum circuit, however, naïve translations typically require a vast amount of ancillary registers.

For notational convenience, we will solve the following equivalent problem.

Problem 2. A system of quadratic equations over $\mathbb{F}_2$ in **convenient form** is given by a 'cube' $(\lambda_{ij}^{(k)})_{i,j,k}$ in $\mathbb{F}_2$ where $\lambda_{ij}^{(k)} = 0$ whenever $i > j$. The goal is to find $x_1, \ldots, x_n \in \mathbb{F}_2$ such that

$$\sum_{1 \leq i \leq j \leq n} \lambda_{ij}^{(1)} x_i x_j = 1 \quad \ldots \quad \sum_{1 \leq i \leq j \leq n} \lambda_{ij}^{(m)} x_i x_j = 1.$$

Clearly every system in convenient form is also a regular system. Now we describe how to turn any system $(\lambda_{i,j}^{(k)}), (v_k)$ of m equations in n variables into an equivalent system $(\lambda'^{(k)}_{i,j})$ of $m+1$ equations in $n+1$ variables that is in convenient form. For $1 \leq i, j \leq n+1$ and $1 \leq k \leq m$ define

$$\lambda'^{(k)}_{i,j} := \begin{cases} \lambda_{i,j}^{(k)} & i = j \leq n \\ \lambda_{i,j}^{(k)} + \lambda_{j,i}^{(k)} & i < j \leq n \\ 1 + v_k & i = j = n+1 \\ 0 & \text{otherwise} \end{cases} \qquad \lambda'^{(m+1)}_{i,j} := \begin{cases} 1 & i = j = n+1 \\ 0 & \text{otherwise.} \end{cases}$$

The new equation forces $x_{n+1} = 1$ and so the new terms $\lambda'^{(k)}_{n+1,n+1} = 1 + v_k$ compensate for having a constant term 1.

The first oracle we construct, will use at most $n + m + 2$ qubit-registers and $O(mn^2)$ time for a system of m quadratic equations in convenient form with n variables. Our second oracle will only use $n+3$ qubit-registers, but require approximately double the amount of time.

To conveniently describe our circuit later on, define

$$y_i^{(k)} = \sum_{1 \leq j \leq n} \lambda_{ij}^{(k)} x_j \qquad E^{(k)} = \sum_{1 \leq i \leq n} x_i y_i^{(k)}.$$

Then (x_i) is a solution if and only if $E^{(k)} = 1$ for every $1 \leq k \leq m$.

Example 1. As a running example throughout the paper, we will use the following small system:

$$\begin{aligned} x_1(1 + x_2 + x_3) + x_2 x_3 &= 1 \\ x_2(1 + x_3) &= 1 \end{aligned}$$

Before we continue with a step-by-step definition of the circuit for the first oracle, we will review with the basics of quantum computing and in particular Grover's algorithm.

2.2 Quantum computing

We start with finite classical computing and describe finite quantum computing later in a similar fashion. Write $\underline{n}$ for the set of natural numbers less than n. Clearly $\underline{2^n}$ is the set of possible states of an n-bit unsigned integer. Classically every function from $f\colon \underline{2^n} \to \underline{2^m}$ is computable. However, some are easy to compute and others are practically infeasible. One measure of complexity is the size of the smallest Boolean circuit containing just NAND-gates that computes f.

Later we will see that it is not easy for a quantum computer to efficiently compute any classical function f, because every quantum gate must be invertible. For every classical simple reversible gate, however, there exists a counterpart quantum gate. In the construction of our oracles we will only use (the quantum counterparts of) classical reversible gates.

The state of a quantum computer with n qubits is a tuple $(a_0, \ldots, a_{2^n-1})$ of 2^n complex numbers with $|a_0|^2 + \cdots + |a_{2^n-1}|^2 = 1$. It is convenient to write subscripts of a in binary, e.g. $a_{1\ldots1} := a_{2^n-1}$. If one opens up the quantum computer and looks at the qubits, one will find that they *collapse* into some classical state of just n bits in a non-deterministic fashion. The chance to find all qubits in the classical state 0 is $|a_{0\ldots0}|^2$. Similarly $|a_{b_1\ldots b_n}|^2$ is the chance to find the first qubit as the classical bit b_1, the second qubit as the classical bit b_2 and so on.

It is customary to define $|b_1 \ldots b_n\rangle$ to be the state which is zero everywhere except for on the $b_1 \ldots b_n^{\text{th}}$ place. For example $|0 \ldots 0\rangle = (1, 0, \ldots, 0)$, $|1 \ldots 1\rangle = (0, \ldots, 0, 1)$ and $\frac{|00\rangle+|11\rangle}{\sqrt{2}} = (\frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}})$. This last state is interesting: if one measures the first qubit to be 0 (resp. 1), one is sure that the second qubit must be 0 (resp. 1) as well. The two qubits are said to be *entangled*.

Every unitary complex $2^n \times 2^n$ matrix U preserves length and thus will send a state a to a new state Ua. Every operation a quantum computer can perform (except for measurement) will be of this form. Conversely, every unitary (matrix) is realizable by a universal quantum computer.

However, just like in the classical case, not every unitary is efficient to compute. It is not yet clear what the primitive operations of the first practical quantum computer will be and thus what would be the appropriate basic gates of this quantum computer — or whether gate-count itself would be the most apt measure of complexity. For instance, some gates (the Toffoli gates) in the gate set we will use are more costly to make fault tolerant with the current quantum error correcting codes than the others. For now we will make do.

If $f\colon \underline{2^n} \to \underline{2^n}$ is a reversible map, there is a unitary U_f fixed by $U_f\,|b_1 \ldots b_n\rangle = |f(b_1 \ldots b_n)\rangle$. In this way a *reversible* function corresponds to a quantum program.

2.3 Applying Grover's algorithm

Problem 3. Let $f\colon \underline{2^n} \to \underline{2}$ be a function which is valued 0 everywhere except on one place. The problem is to find, given a Boolean circuit for f, the place where f is valued 1.

Classically one cannot do better in general than to try every possible input. On average one will have to execute f for 2^{n-1} times. With a quantum computer this problem can be solved with high probability by executing the quantum analogue of f just $2^{\frac{1}{2}n}$ times using just n qubits. This is done using Grover's algorithm. Actually, Grover's algorithm (with the quantum counting extension [BHT98]) solves the more general problem where f has arbitrarily many places where it is valued 1 and one is interested in any preimage of 1. In this paper, however, we only need the simpler version.

Clearly f is not reversible. How can we then feed it to a quantum computer where every operation should be invertible? One way is to define a new classical

reversible function $R_f \colon \underline{2^{n+1}} \to \underline{2^{n+1}}$ by

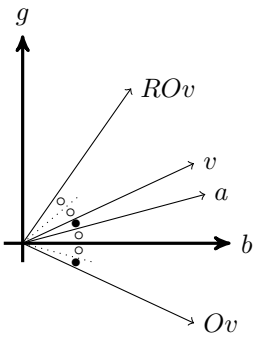
$$R_f(b_1 \dots b_n y) = \begin{cases} b_1 \dots b_n y & f(b_1 \dots b_n) = 0, \\ b_1 \dots b_n \overline{y} & f(b_1 \dots b_n) = 1. \end{cases}$$

Here overline denotes negation. For Grover's algorithm it is sufficient to provide a quantum circuit, the *oracle*, which is the quantum analogue of R_f. We claimed Grover only needs n qubits. This is true, however in practice the oracle might not be efficient to compute with just n qubits. Often the oracle itself requires some ancillary qubits, say m, as scratch space to be efficient. In that case Grover uses a total of $n + m$ qubits.

The gist of Grover's algorithm. To understand the remainder of this paper, it is not required to know how Grover's algorithm works (if the reader accepts that the core part are evaluations of the oracle). However, for completeness, we provide a brief summary of Grover's algorithm.

Let $f \colon \underline{2^n} \to \underline{2}$ be any function for which we want to find a $w \in \underline{2^n}$ with $f(w) = 1$. Write a, g, b respectively for the standard uniform superposition of all basisvectors, the basisvectors marked 1 by f, and the basisvectors marked 0 by f. Concretely, with $N = 2^n$ and $M = |f^{-1}(1)|$:

$$a = \sum_{w \in \underline{2^n}} \frac{1}{\sqrt{N}} |w\rangle \qquad g = \sum_{\substack{w \in \underline{2^n} \\ f(w)=1}} \frac{1}{\sqrt{M}} |w\rangle \qquad b = \sum_{\substack{w \in \underline{2^n} \\ f(w)=0}} \frac{1}{\sqrt{N-M}} |w\rangle$$

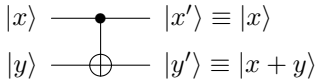


If we can put the quantum computer in state g, then a measurement will give a bitstring w with $f(w) = 1$ as desired. It is easy to see that a is actually a linear combination of b and g: $a = \frac{\sqrt{M}}{\sqrt{N}} g + \frac{\sqrt{N-M}}{\sqrt{N}} b$. As b and g are orthogonal, we can visualize a as a point on a grid with axes g and b. Let O be the unitary with $O|w\rangle = |w\rangle$ if $f(w) = 0$ and $O|w\rangle = -|w\rangle$ if $f(w) = 1$. It is not hard to construct O from the oracle discussed above (the quantum analogue of R_f). In our picture, O is simply a reflection over the b axis. Note how an arbitrary v on the grid is reflected to Ov. Let R denote the unitary that reflects over a. By adding some angles in the picture and a moments thought, one can see the action of RO is a counter-clockwise rotation in our grid by twice the angle a has with b. If M is known, this angle is straightforward to compute. Grover's algorithm is to prepare the quantum computer in state a and then to execute as many times the unitary RO until the state of the computer is close to g. Measuring the bits will then give a bitstring w with $f(w) = 1$ with high probability. The number of times that RO has to be executed can be shown [NC10, Eq. 6.17] to be at most $\lceil \frac{\pi}{4} \sqrt{N/M} \rceil$.

3 A collection of quantum gates

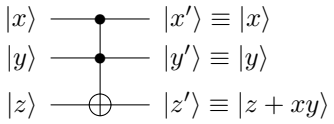
In this section we collect the quantum gates that we will use for the oracles presented in Sections 4 and 5. All quantum gates we will use are the quantum counterparts of reversible classical gates.

Gate 1 *We will use a* ***CNOT gate*** *(controlled not — also called the Feynman gate) to compute XOR. CNOT is usually drawn as shown below on the left. As unitary it is defined on the computational basis by* $\mathrm{CNOT}\,|x\rangle\,|y\rangle = |x\rangle\,|x+y\rangle$*. It corresponds to the classical reversible Boolean function on the right.*



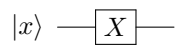
x	y	x'	y'
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

Gate 2 *To compute AND, we will use the* ***Toffoli gate*** T*. It's drawn below on the left. As unitary it is defined by* $\mathrm{T}\,|x\rangle\,|y\rangle\,|z\rangle = |x\rangle\,|y\rangle\,|z+xy\rangle$ *(on the computation basis). It corresponds to the classical invertible Boolean function on the right.*



x	y	z	x'	y'	z'
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Gate 3 *To compute NOT, we use the* X***-gate****, usually depicted by*

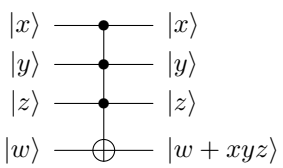


As unitary it is defined by $X\,|x\rangle = |\overline{x}\rangle = |1+x\rangle$ *(on the computational basis).*

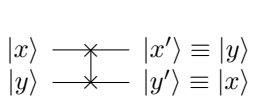
Gate 4 *To compute the AND of multiple bits, we will use the* n***-qubit Toffoli gate*** *(*T_n*). It is a controlled not-gate with* $n-1$ *control-bits. That is: its action as a unitary on the computational basis is*

$$T_n\,|x_1\rangle \cdots |x_n\rangle = |x_1\rangle \cdots |x_{n-1}\rangle\,|x_n + (x_1 \cdot \cdots \cdot x_{n-1})\rangle\,.$$

Note that $T_1 = X$*,* $T_2 = \mathrm{CNOT}$ *and* $T_3 = T$*. The* n*-qubit Toffoli gate is drawn similarly to the regular Toffoli gate. For instance, this is the 4-qubit Toffoli gate:*



Gate 5 *For the second oracle we want to swap bits, which is done with the* 2*-qubit* ***swap-gate*** *S.It's drawn below on the left*[2]*. As a unitary it is defined by* $S|x\rangle|y\rangle = |y\rangle|x\rangle$ *(on the computational basis). It corresponds to the classical invertible Boolean function on the right.*

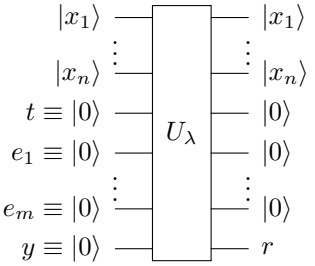


x	y	x'	y'
0	0	0	0
0	1	1	0
1	0	0	1
1	1	1	1

It is expected that the X, SWAP and CNOT gates will be cheap to execute and error correct on a quantum computer, whereas (n-qubit) Toffoli gates will be expensive. This is why papers often list gate-counts separately for 'easy' and 'hard' gates.

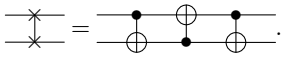
4 The first Grover oracle for $\mathcal{MQ}$ over $\mathbb{F}_2$

Our circuit U_λ to check whether $(x_i)_i$ is a solution (of a system of m quadratic equations in n variables in convenient form), will use $n+m+2$ registers. It will act as follows, where $r = |1\rangle$ if $(x_i)_i$ is a solution and $|0\rangle$ else.

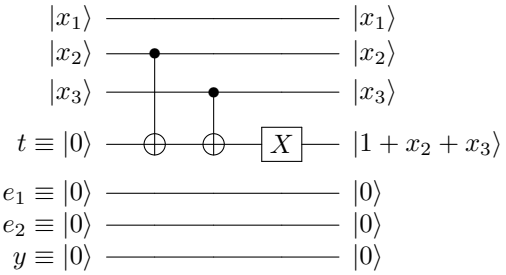


The first n registers are the input and should be initialized with $x_1, \ldots, x_n$. The circuit will not change them – not even temporarily. The next register will be an ancillary register labelled t. It is intended to be initialized to $|0\rangle$. The next m registers we will label e_1, ..., e_m and should all be initialized to $|0\rangle$. The final register is an output register labelled y.

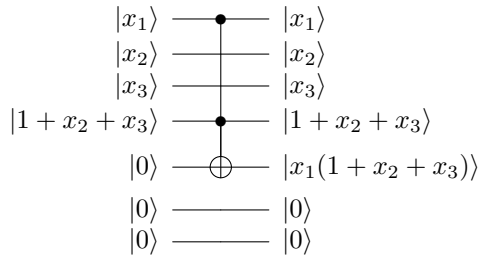
We will construct our circuit U_λ step by step. Note that $1 + z = \overline{z}$. Thus, with at most $n-1$ CNOT gates and possibly an X-gate, we can put $y_1^{(1)}$ into t.

[2] Note that a SWAP gate can be written with CNOTs:  .

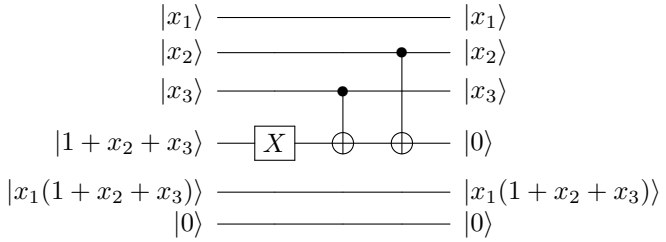
In our example (see Section 2):



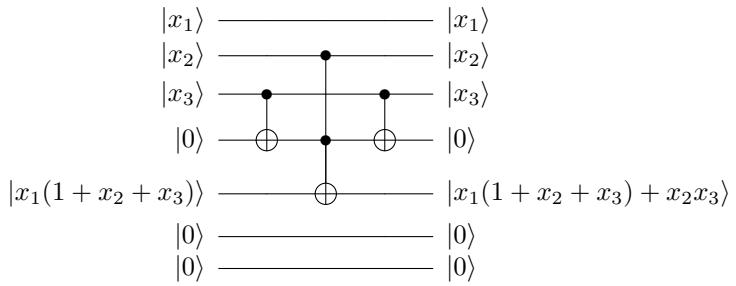
Using one Toffoli gate, we put $x_1 y_1^{(1)}$ into e_1. In our example:



Then, by applying the inverse circuit used to put $y_1^{(1)}$ into t, we can return t to $|0\rangle$. As all the gates we use are self-inverse, the inverse circuit is simply the horizontal mirror-image. In our example:

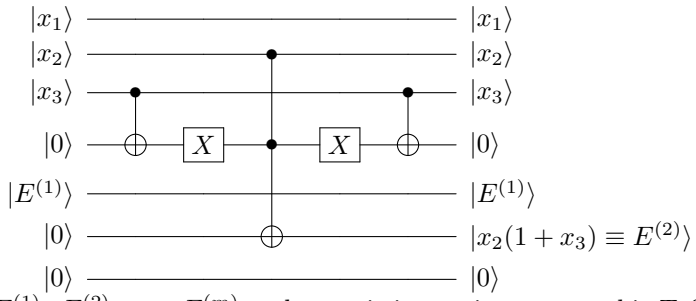


Using a similar circuit with at most $2n - 4$ CNOT-gates, two X-gate and a Toffoli-gate, we can add $y_2^{(1)}$ to e_1, leaving the remaining registers untouched. In our example $y_2^{(1)} = x_2 x_3$, hence we obtain the following:



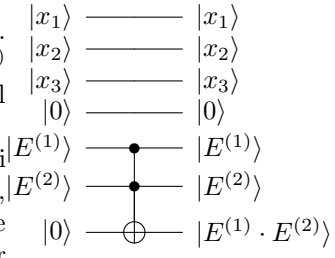
We continue with $n-2$ similar circuits, to add $y_2^{(1)}$, ..., $y_n^{(1)}$ to e_1. Our complete circuit up to this point, has put $E^{(1)}$ into e_1 with at most n^2+2n gates. (In our example we are already done.) The remaining registers are as they were.

With $m-1$ similar circuits we can store $E^{(k)}$ into e_k for the other k. In total we will have used at most $m(n^2+2n)$ gates. In our example the remainder will be:



Next, compute $E^{(1)} \cdot E^{(2)} \cdot \cdots \cdot E^{(m)}$ and store it in y using an m-qubit Toffoli gate.

The circuit for our example is shown on the right. Finally, we reverse the computation of $E^{(1)}$, ..., $E^{(m)}$ to reset all but the output register to their initial state. We have used at most $2m(n^2+2n)+1$ gates.



One might object to counting the n-qubit Toffoli gate with the same weight as the other gates. Indeed, classically one cannot even compute arbitrarily large reversible circuits if one is restricted to m-register gates and no temporary storage [Tof80, Thm. 5.2]. However, without ancillary qubits and just with CNOTs and one-qubit gates, one can create an n-qubit Toffoli gate. If one allows one ancillary qubit, one only needs $O(n)$ many ≤ 2-qubit gates to construct a n-qubit Tofolli gate [MD03]. The gates used in this construction are, however, expensive to error correct with current codes. For the next oracle, we will implicitly construct a 2^n-qubit Toffoli gate from an n-qubit Toffoli gate with n ancillary qubits.

Python and Quipper code to generate the oracle presented in this section are given in Appendix A.

5 The second Grover oracle for $\mathcal{MQ}$ over $\mathbb{F}_2$

In this section we will describe a second, more complex oracle, which requires fewer qubits, but approximately twice the number of gates. As for the first oracle, we give Quipper code to generate this second oracle in Appendix A.

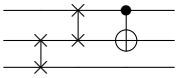
In our first oracle we reserved for every equation a qubit register which stores whether that equation is satisfied. At the end the oracle checks whether every equation is satisfied by checking whether every of the corresponding registers is set to $|1\rangle$. Instead, for our second oracle we will only count the number of equations that are satisfied. Instead of m separate registers, we will only need $\lceil \log_2 m \rceil$

registers which act as a counter. Instead of storing $E^{(k)}$ into a separate register, the oracle will do a controlled increment on the counter. At the end the oracle will check whether the value in the counter is m. This can be done with suitably placed X-gates and a multi-qubit Toffoli.

Note that as the value of $E^{(k)}$ is not kept around anymore, it needs to be computed and uncomputed a second time compared to the first oracle to uncompute the counter qubits. This is the reason the second oracle requires approximately double the number of gates.

We still have to describe the increment circuit for the counter register. Using the standard binary encoding for the counter and the obvious increment is not a good a choice: the incrementation is hard to implement efficiently without using ancillary registers. We can do better by not adhering to the standard binary encoding.

For instance consider the 3-qubit circuit on the right. This circuit has two (classical) cycles: it will send $|000\rangle$ directly to $|000\rangle$ and

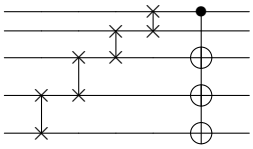


$$|111\rangle \mapsto |101\rangle \mapsto |100\rangle \mapsto |010\rangle \mapsto |001\rangle \mapsto |110\rangle \mapsto |011\rangle \mapsto |111\rangle\,.$$

This simple 3-qubit circuit can thus be used as a counter up to 7. For instance, to count 5 equations with this circuit one initializes the counter register to $|100\rangle$, applies the circuit for each valid equation and checks in the end whether the counter register is set to $|111\rangle$.

Now we will show how to construct a similar simple circuit for any number of qubits. For this construction we will need to think of the state $|v_1 \dots v_n\rangle$ as the polynomial $v_n x^{n-1} + \cdots + v_2 x + v_1$ over $\mathbb{F}_2$. For instance $|1101\rangle$ corresponds to $1 + x^2 + x^3$. The circuit above corresponds to multiplying by x in the field $\mathbb{F}_2[x]/(x^3 + x + 1)$. Indeed: the ladder of swap gates at the start of the circuit is a rotation down and would correspond to multiplying by x in the ring $\mathbb{F}_2[x]/(x^3 + 1)$. The cNOT at the end of the circuit is responsible for the missing x term. The fact that the circuit cycles over all (7) invertible elements of the field is by definition equivalent to the fact that $x^3 + x + 1$ is a primitive polynomial.

So, to construct a counter on c-qubits, one picks a primitive polynomial $p(x)$ over $\mathbb{F}_2$ of degree c (eg. from [Wat62]) and builds the corresponding circuit. For instance, $x^5 + x^4 + x^3 + x^2 + 1$ is a primitive polynomial and corresponds to the circuit on the right.



The following table lists the maximum number of each gate used in the second oracle compared to the first for a system of 81 equations in 85 variables.

	qubits	X	CNOT	Toffoli	*and*
First oracle	168	27,710	1,156,680	13,770	*one* 81-Toffoli
Second oracle	94	55,250	2,316,276	27,702	*one* 7-Toffoli

To find a solution to this example system, the oracle will be executed $\sim 2^{40}$ times interleaved with reflections, which yields a total of $\sim 2^{61}$ executed gates when using the second oracle.

6 Circuit depth

If gates act on separate qubits, they might be executed in parallel. For this reason the depth of a circuit is often considered instead of the total number of gates executed. For our first two oracles we choose to optimize for qubit count instead of circuit depth. We will briefly sketch how to decrease the circuit depth by allowing for more qubit registers.

For simplicity we will assume that CNOTs an Tofolli's with different target wires (but possibly the same control wires) can be executed in parallel. If one changes the first oracle to use a separate t register for each equation, the value of each equation can be computed practically in parallel and the circuit depth is reduced from $O(n^2m)$ to $O(n^2 + m)$ using a total of $n + 2m + 1$ registers.

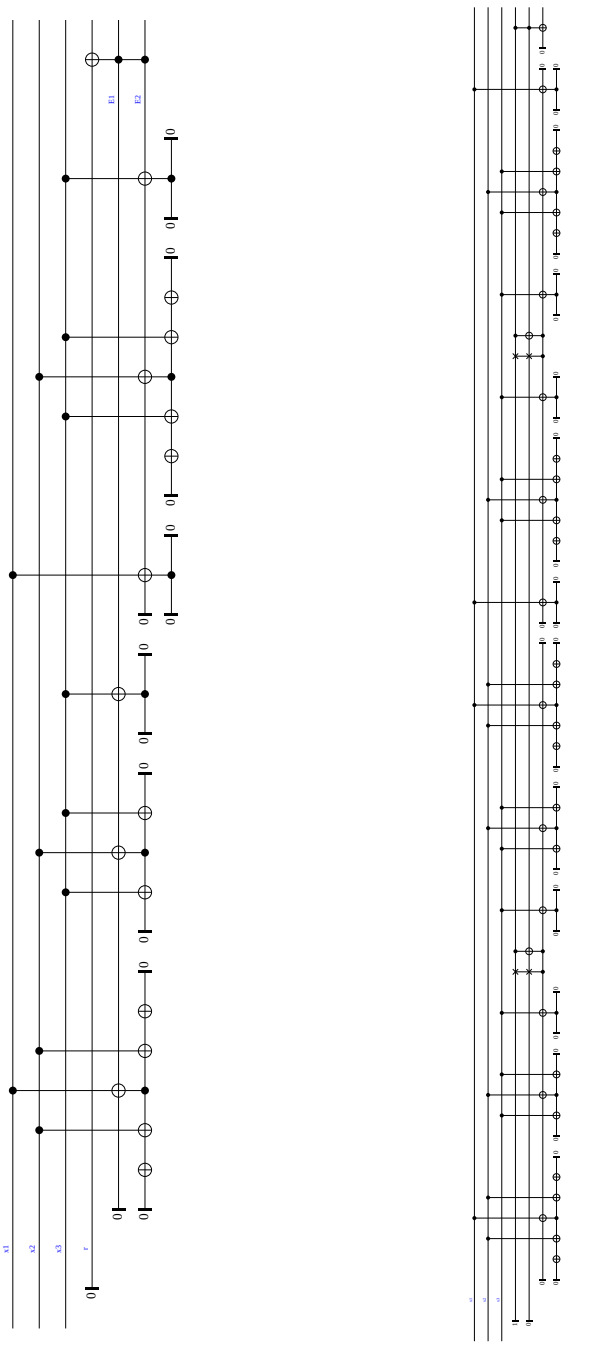
There is still room for another trade-off: the terms $y_i^{(k)}$ for a single equation are not computed in parallel. If one uses a separate register for each $y_i^{(k)}$, one could reduce the circuit depth to $O(n + m)$ using a total of $n + nm + m + 1$ registers.

7 Conclusion

We have shown step-by-step how to construct oracles for Grover's algorithm to solve binary $\mathcal{MQ}$, implement these in a quantum programming language, and estimate the resources it will use. As a corollary we find that some proposed choice of parameters for some "post-quantum" schemes seem practical to break on a quantum computer with less than a hundred logical qubits.

We finish with a table that shows the upper bound of resources required to solve a system with 84 equations in 80 variables with a single solution. (This is the hard problem underlying the identification scheme described in [SSH11] for "80-bits security".) We reiterate that the number of gates mentioned is the cumulative number of times that kind is executed.

	first oracle	second oracle
qubits	168	94
X gates	13,5363,390,216,963,920	269,896,330,187,198,000
CNOT gates	5,650,383,478,749,831,360	11,300,766,957,499,662,720
Toffoli gates	67,266,469,985,117,040	135,324,310,205,353,104
7-qubit Toffoli gates	0	9,770,002,902,704
81-qubit Toffoli gates	9,770,002,902,704	0
84-qubit Toffoli gates	4,885,001,451,352	4,885,001,451,352
Hadamard gates	840,220,249,632,629	200,285,059,505,513
Controlled swap	0	4,748,221,410,714,144
Controlled-Z gates	4,885,001,451,352	4,885,001,451,352
Total number of gates	5,989,207,179,415,606,250	11,982,322,359,992,293,930



(a) First oracle (b) Second oracle

Fig. 1: Oracles for the running example generated by Quipper

Acknowledgments

The authors are grateful to Gauillaume Allais and Peter Selinger for their helpful suggestions. In particular, it was Peter Selinger's suggestion to construct a counter from a primitive polynomial. Ben Pring spotted two errors in §6.

References

AMG+16. Matthew Amy, Olivia Di Matteo, Vlad Gheorghiu, Michele Mosca, Alex Parent, and John Schanck. Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3. Preprint arXiv:1603.09383, 2016. https://arxiv.org/abs/1603.09383. 2

BCC+14. Charles Bouillaguet, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. Fast exhaustive search for quadratic systems in $\mathbb{F}_2$ on FPGAs. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*, volume 8282 of *LNCS*, pages 205–222. Springer, 2014. http://polycephaly.org/papers/#forcemq. 3

BHT98. Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum counting. In *Automata, Languages and Programming*, pages 820–831. Springer, 1998. 3, 5

Chu05. Isaac Chuang. Quantum circuit viewer: qasm2circ, 2005. http://www.media.mit.edu/quanta/qasm2circ/ (accessed 2016-06-24). 16

DS05. Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security*, volume 3531 of *LNCS*, pages 164–175. Springer, 2005. 2

GJ79. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979. 2

GLR+13a. Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. An introduction to quantum programming in quipper. In Gerhard W. Dueck and D. Michael Miller, editors, *Reversible Computation*, volume 7948 of *LNCS*, pages 110–124. Springer, 2013. 16

GLR+13b. Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. 48(6):333–342, 2013. https://arxiv.org/pdf/1304.3390. 16

GLRS16. Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying Grover's algorithm to AES: quantum resource estimates. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography*, volume 9606 of *LNCS*, pages 29–43. Springer, 2016. 2

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*, pages 212–219. ACM, 1996. 1, 3

KPG99. Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *Advanced in Cryptology – EUROCRYPT '99*, volume 1592 of *LNCS*, pages 206–222. Springer, 1999. extended version at http://andrewl.dreamhosters.com/archive/45339168.pdf. 2

MD03. Dmitri Maslov and Gerhard W Dueck. Improved quantum cost for n-bit Toffoli gates. *Electronics Letters*, 39(25):1790–1791, 2003. 10

NC10. Michael A. Nielsen and Isaak L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2010. 2, 3, 6

PCG01. Jacques Patarin, Nicolas Courtois, and Louis Goubin. QUARTZ, 128-bit long digital signatures. In David Naccache, editor, *Topics in Cryptology – CT-RSA 2001*, volume 2020 of *LNCS*, pages 282–297. Springer, 2001. 2

PCY+15. Albrecht Petzoldt, Ming-Shing Chen, Bo-Yin Yang, Chengdong Tao, and Jintai Ding. Design principles for HFEv- based multivariate signature schemes. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, volume 9452 of *LNCS*, pages 311–334. Springer, 2015. http://www.iis.sinica.edu.tw/papers/byyang/19342-F.pdf. 2

Sel. Peter Selinger. The quipper language. http://www.mathstat.dal.ca/~selinger/quipper/ (accessed 2016-09-01). 16

Sho94. Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *SFCS '94 Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE, 1994. http://www-math.mit.edu/~shor/papers/algsfqc-dlf.pdf. 1

Sho97. Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26:1484–1509, 1997. http://arxiv.org/abs/quant-ph/9508027. 1

SSH11. Koichi Sakumoto, Taizo Shirai, and Harunaga Hiwatari. Public-key identification schemes based on multivariate quadratic polynomials. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *LNCS*, pages 706–723. Springer, 2011. https://www.iacr.org/archive/crypto2011/68410703/68410703.pdf. 2, 3, 12

Tof80. Tommaso Toffoli. *Reversible computing*. Springer, 1980. 10

Wat62. E.J. Watson. Primitive polynomials (mod 2). *Mathematics of Computation*, 16(79):368–369, 1962. 11

YCC04. Bo-Yin Yang, Jiun-Ming Chen, and Nicolas T. Courtois. On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis. In Eiji Okamoto Javier Lopez, Sihan Qing, editor, *Information and Communications Security*, volume 3269 of *LNCS*, pages 401–413. Springer, 2004. http://www.iis.sinica.edu.tw/papers/byyang/2384-F.pdf. 3

A Example code

The following is Python code that generates the first oracle circuit, which we described informally in Section 4.

```
def create_circuit(n, m, sqe):
    """ Creates Circuit for Grover oracle that solves the system of
        quadratic equations sqe over F_2 in standard form

            n: number of variables x_i in sqe
            m: number of equations in sqe
            sqe[k][i][j]: true if x_ix_j occurs in the k-th equation. """
    # First, create helper circuit that puts E^(k) into e_k
    E_circuit = Circuit()
    for k in range(1, m+1):
```

```
        for i in range(1, n+1):
            if not any(sqe[k-1][i-1]): continue
            # Another helper circuit, that XORs y_i^(k) into t
            y_circuit = Circuit()
            for j in range(i+1, n+1):
                if sqe[k-1][i-1][j-1]:
                    y_circuit.CNOT('x'+str(j), 't')
            if sqe[k-1][i-1][i-1]:
                y_circuit.X('t')

            # XOR the value (x_i AND  y_i^(k)) into e_k
            # and clear t afterwards
            E_circuit.extend(y_circuit)  # first put y_i^(k) into t
            E_circuit.toffoli('x'+str(i), 't', 'e'+str(k))
            E_circuit.extend(y_circuit.inverse())  # uncompute t

    # Now, assemble the whole circuit
    circuit = Circuit()
    circuit.extend(E_circuit)  # put E^(k) into e_k
    # put result into y
    circuit.add('toffoli{0}'.format(m),
                    ['e{0}'.format(i) for i in range(1,m+1)] + ['y'])
    circuit.extend(E_circuit.inverse())  # uncompute e_k
    return circuit
```

To turn this into a useful commandline util that converts a system of quadratic equations into a quantum circuit in Nielsen and Chuang's QASM[Chu05] format, we need a few more lines of code.[3] One one invokes the completed script as follows.

```
python mqgrover.py 3 2 111010000110
```

The second oracle is more complex and easier to synthesize in a special purpose language. The following is an implementation of the first and second oracle in the quipper programming language[GLR+13b,GLR+13a,Sel], which is based on Haskell.

```
module MQGrover (oracle1, oracle2) where

import Quipper
import Data.Bits
import Data.List
import Control.Monad
import Control.Applicative
```

[3] https://github.com/bwesterb/mqgrover

```
--
-- First oracle
--

-- Compute y_i^(k) from the coefficients lambda_ii^(k), ..., lambda_in^(k)
-- and the assignment x_1, ..., x_n.
compute_y :: [Bool] -> [Qubit] -> Circ (Qubit)
compute_y cs xs = withM (qinit False) $ \t ->
  unless (null cs) $ do
    when (head cs) $ qnot_at t
    zipWithM_ (\c x ->  when c (qnot_at t `controlled` x)) (tail cs) (tail xs)

-- Computes E^(k) from the "triangle" lambda^(k)_ij (1 <= j <= n)
-- and the assignment x_1, ..., x_n.
compute_e :: [Qubit] -> [[Bool]] -> Circ (Qubit)
compute_e xs css =
  withM (qinit False) $ \e ->
    forM_ (zip css (init $ tails xs))  $ \(cs, xs') ->
      with_computed (compute_y cs xs') $ \t ->
        qnot_at e `controlled` (t, head xs')

-- The first (straight-forward) oracle.  Computes whether the
-- assignment x_1, ..., x_n satisfies the given system of equations.
oracle1 :: [[[Bool]]] -> [Qubit] -> Circ (Qubit)
oracle1 csss xs =
  withM (qinit False) $ \r -> do
    label (r:xs) ("r":["x" ++ (show i) | i <- [1..length xs]])
    es <- mapM (compute_e xs) csss
    label es ["E" ++ (show i) | i <- [1..length es]]
    qnot_at r `controlled` es

--
-- Second oracle that uses only n + ceil(log_2 m) + 3 registers, but requires
-- more gates.
--

oracle2 :: [[[Bool]]] -> [Qubit] -> Circ (Qubit)
oracle2 csss xs = do
  ctr <- init_counter $ length csss
  label xs ["x" ++ (show i) | i <- [1..length xs]]
  forM_ csss $ \css ->
    with_computed (compute_e xs css) $ controlled $ inc_counter ctr
  check_counter ctr

--
```

```haskell
-- Helpers for the second oracle.
--

-- Rotates qubits around one turn
qrotate :: [Qubit] -> Circ()
qrotate qs = zipWithM_ swap qs' $ tail qs' where qs' = reverse qs

-- Turns polynomial into corresponding circuit
apply_polynomial :: [Bool] -> [Qubit] -> Circ()
apply_polynomial cs qs = do
  qrotate qs
  zipWithM_ (\c q -> do
        when c $ qnot_at q `controlled` head qs) (init $ tail cs) (tail qs)
  return ()

-- Returns number of bits in the binary expansion of a given integer
bits_required :: Int -> Int
bits_required 0 = 0
bits_required n = 1 + bits_required (n `shiftR` 1)

type QCounter = (Int, [Qubit])
bound :: QCounter -> Int
bound (n, qs) = n
qbits :: QCounter -> [Qubit]
qbits (n, qs) = qs

-- Creates a new counter to count up to the given integer.
init_counter :: Int -> Circ(QCounter)
init_counter n = do
  qs <- qinit $ iterate (class_inc_counter zero_poly) (replicate nbits True)
                !!  (2^nbits - n)
  return (n, qs)
    where
      nbits = bits_required n
      zero_poly = prim_poly nbits
      class_inc_counter :: [Bool] -> [Bool] -> [Bool]
      class_inc_counter zero_poly (False:cs) = cs ++ [False]
      class_inc_counter zero_poly (True:cs)
        = zipWith xor (cs ++ [False]) (tail zero_poly)

-- Increment the counter by one.
inc_counter :: QCounter -> Circ ()
inc_counter (n,qs) = prim_poly (bits_required n) `apply_polynomial` qs

-- Check whether the counter has reached the desired value
```

```
check_counter :: QCounter -> Circ (Qubit)
check_counter qc = withM (qinit False) $ \t -> qnot_at t `controlled` qbits qc

-- Returns a primitive polynomial over F_2 of given degree
prim_poly :: Int -> [Bool]
prim_poly n = map (`elem` 0:n:(watson_prim_polies!!n)) [0..n]

-- Contains for each n <= 32  a primitive polynomial of order n modulo F_2.
-- The number are the non-trivial powers of x that occur: for instance
-- the list [4,3,2] at index 8 represents the primitive polynomial
--      x^8 + x^4 + x^3 + x^2 + 1.
-- List taken from E. J. Watson, 1961.
watson_prim_polies = [
  [], [], [1], [1], [1], [2], [1], [1], [4, 3, 2], [4], [3], [2], [6,
  4, 1], [4, 3, 1], [5, 3, 1], [1], [5, 3, 2], [3], [5, 2, 1], [5,
  2, 1], [3], [2], [1], [5], [4, 3, 1], [3], [6, 2, 1], [5, 2, 1],
  [3], [2], [6, 4, 1], [3], [7, 5, 3, 2]]

withM :: Monad m =>  m a -> (a -> m ()) -> m a
withM f g = do
  t <- f
  g t
  return t
```

The gate counts mentioned in the conclusion were generated by the build-in `GateCount` functionality of Quipper, which was invoked (for the first oracle) with the following code.

```
print_simple GateCount $ grover (oracle1 sqe) 85 1
```

The variable `sqe` is set to the system of 81 equations in 85 variables where every coefficient is 1 as it requires most gates executed in our construction. We use the following implementation of Grover's algorithm.

```
module Grover (grover) where

import Quipper

import Control.Monad

-- |Reflects (in place) over the standard uniform superposition of qubits.
-- This is used as the second part of the Grover iteration.
reflect_over_a :: [Qubit] -> Circ ()
reflect_over_a qs = do
  with_basis_change (forM_ qs hadamard_at) $ do
    with_basis_change (forM_ qs qnot_at) $ do
      with_basis_change (hadamard_at $ head qs) $ do
```

```
        qnot_at (head qs) `controlled` tail qs

-- |Grover's algorithm.  'oracle' is a circuit that should map exactly
-- 'm' 'n'-qubit-words to |1> and the others to |0>.  'grover' returns
-- a circuit that creates a superposition over all 'n'-qubit words that
-- are mapped to |1> by the oracle.
grover :: ([Qubit] -> Circ (Qubit)) -> Int -> Int -> Circ ()
grover oracle n m = do
  -- Create a standard uniform superposition over all qubits.
  qs <- qinit $ replicate n False
  forM_ qs hadamard_at
  iterations qs
  return ()
    where
      n_iters = floor $ pi / 4 / asin(sqrt((fromIntegral m) / (2^n)))
      iterations :: [Qubit] -> Circ ([Qubit])
      iterations = nbox "grover-iteration" n_iters $ \qs -> do
        -- First, the adapted oracle.  The following will send a computational
        -- basis vector w to -w if its tagged by the original oracle and
        -- leave it in place otherwise.
        with_computed (oracle qs) $ \r -> do
          gate_Z_at (head qs) `controlled` r
        -- Then, reflect over the standard uniform superposition.
        reflect_over_a qs
        return qs
```