Simulating Attacks on Authenticated Semi-Quantum Direct Communication Protocol

Nathan Marshall
Ottawa, Canada
nathsmar@gmail.com

Nick Rivard
Ottawa, Canada
nhrivard@gmail.com

***Abstract*—This study presents a detailed analysis of the randomization-based authenticated semi-quantum direct communication protocol presented by Yi-Ping Luo and Tzonelih Hwang, as well as several of the attacks already examined in their analysis of the protocol. The protocol uses pre-shared keys rather than a classical channel, and requires only the sender to have advanced quantum capabilities. We have implemented a simulation of the protocol and confirmed that it is resistent to impersonation attacks, intercept-and-resend attacks, and one-qubit modification attacks. We have also solved the problem of reordering qubits into a random permutation based on a key, which was not explained in the original description of the protocol.**

***Index Terms*—Authentication, Authenticated semi-quantum communication, Bell states, Quantum communication, Quantum cryptography, Semi-quantum communication**

I. Introduction

While practical quantum communication is feasible using modern technology, the cost of nodes capable of quantum computation will likely remain high for some time. Since simpler, "classical" nodes (with only the ability to read individual qubits) are cheaper, semi-quantum communication protocols are useful for practical, cost-efficient, secure communication.

This work is based on an authenticated semi-quantum direct communication (ASQDC) protocol proposed by Yi-Ping Luo and Tzonelih Hwang that takes advantage of this idea of a "classical" receiver Bob [1]. The present work describes their article and the insights, results, and conclusions gained from implementing the randomization-based version of the ASQDC protcol described therein.

We implemented a simulation of the aforementioned protocol in Matlab using Quantum Information Toolkit, which brought to light parts of the protocol that were not defined thoroughly enough to trivially follow. Our contributions are a useful reference for implementing a real ASQDC network, since the challenges we encountered in reordering qubits based on a shared key would inevitably be encountered by others as well.

We successfully implemented the randomization-based ASQDC protocol as per the specifications laid out in their article, and also generated data concerning the security of the protocol in the presence of four attacks: impersonation of Alice, impersonation of Bob, intercept-and-resend attack, and modification of 1-qubit attack. We slightly modified the definition of the intercept-and-resend attack from the description given by Luo and Hwang, and we disagreed slightly on the security analysis for the impersonation of Bob. Nevertheless, our results support the claim that the protocol is secure against these attacks, except for impersonating Alice with small message size.

II. Problem Domain

A. Authentication

Message authentication is the process of confirming the origin and data integrity of a given message. This means confirming both that the message did in fact originate from a given sender and that the message was not modified during transit. This is often done using a combination of shared secrets (keys) and cryptographic hash functions. If the message can only be transformed into its corresponding plaintext by a particular key which corresponds to a key which is only in the possession of a known sender, then one can infer that the message must have been sent by the known sender since it must have been encrypted using that key. Similarly for data integrity, if a given plaintext is input into a one-way function (a cryptographic hash function) and produces the observed hash value, then it can be inferred that it is the same message that was originally used by the known sender to produce the hash before they encrypted the message. In reality there can be many complications to the above, but this model is sufficient for the present discussion.

B. Requirements for Authentication in Traditional Protocols

Typically, quantum communications protocols implement message authentication indirectly using authenticated classical (non-quantum) channels and nodes which are capable of performing quantum computation. These restraints limit the viable use-cases for message authentication in quantum communications. Firstly, a dedicated and highly available authenticated classical channel might not be feasible in nodes which are powered by battery. Secondly, while quantum computers exist now, they tend to be expensive and sensitive to environmental conditions. This implies that it will be difficult to build and deploy reliable mobile quantum computers.

C. Authenticated Semi-Quantum Key Distribution

To address the requirement that all nodes be capable of quantum computation, Yu et al. proposed Authenticated Semi-Quantum Key Distribution (ASQKD) protocols [2]. In these protocols, in order to circumvent the requirement for an authenticated classical channel, keys are shared between sender and receiver in advance. Furthermore, at the cost of making quantum communications one-way, the receiver need not be capable of quantum computation themselves. To be precise, Bob (the receiver) only needs to be able to perform the following operations: (1) prepare qubits in the classical basis $\{|0\rangle, |1\rangle\}$, (2) measure individual qubits in the the classical basis, (3) permute qubits, and (4) reflect qubits without measuring them. This is in contrast to Alice (the sender), who need to be able to perform the following operations: (1) prepare any arbitrary quantum states and (2) perform arbitrary quantum measurement.

D. Authenticated Semi-Quantum Direct Communication

The paper by Luo and Hwang which introduced the ASQDC protocols – one of which was implemented in the present work – first introduced usage of Bell states (discussed below) into ASQKD protocols in order to prevent information leakage, thus enabling Alice to determine if Eve (a malicious actor) extracted information from the transmission. The details of how this is accomplished are given below.

E. Bell States

Bell states are the four maximally-entangled states that can be produced using two qubits. When the state of one qubit is measured (reduced to a classical bit), the state of the other qubit is instantly known by the party who measured the first. Quantum entanglement – the name for this phenomenon – is a physical property and is not something that can be logically circumvented. If, for example, two photons (qubits) are entangled into a Bell state and one of the photons is sent to another location, the sender will instantly know the value of the transmitted photon when they measure the remaining photon.

F. Cases Implemented

Five cases are considered: no attacker, Eve impersonates Alice, Eve impersonates Bob, Eve intercepts and re-sends the message, and Eve modifies one qubit of the message.

1) No Attacker: In the case where there is no attacker, it is expected that the protocol will function every time (given that, in the simulation, there are no environmental factors). Alice first computes the hash $h(m)$ of the message m and concatenates them into $M = m||h(m)$. The constraints on the hash function h are that it must output a hash equal in length to the message and it must never produce collisions. Given these constraints, the authors opted to simply flip the value of each bit from the message in the hash. While this would not be secure in the wild, Eve will only be brute-forcing the keys in this simulation. Furthermore, it ensures that the protocol can be used to send any length of message without hash collisions. Next, Alice will produce a sequence of Bell-EPR pairs S such that if $M[i]$=0 then $S[i] = |\phi^+\rangle$. Otherwise, $S[i] = |\psi^-\rangle$. Alice then generates a random sequence of Bell-EPR pairs C in the basis $\{|\phi^+\rangle, |\psi^-\rangle\}$ with the same length as S, labeling the first half C_b and the second half C_a. Next, Alice tensors S and C, producing SC_{ba}. SC_b is then randomly shuffled using key K_1 and sent to Bob as Q. Alice retains C_a.

Upon reception of SC_b, Bob reorders $S'C_b'$ using K_1 and then perfroms a Z-basis measurement on S', obtaining the measurement result MR_B, from which Bob can computer the message and hash $M' = m'||h(m)'$. Bob then computes $h(m')$ and compares it to $h(m)'$ and, if they are not equal, Bob and Alice terminate the protocol.

If $h(m') = h(m)'$, Bob reorders C_b' using K_2 and reflects the result, C_b'' back to Alice.

Alice then applies K_2 to C_b'', obtaining C_b''' and then compares $C_b'''C_a$ with C and, if they are equal, believes that the protocol worked properly.

2) Impersonating Alice: Eve can communicate with Bob, pretending to be Alice, by simply taking on the role of Alice in the protocol described above, using Eve has the requisite quantum computing capabilities described above. The only difference is that Eve needs to guess which keys K_1, K_2 to use.

3) Impersonating Bob: Likewise, Eve can impersonate Bob by receiving the message from Alice and using the part of the protocol normally used by Bob. As above, Eve will need to guess the keys K_1, K_2.

4) Intercept-and-Resend Attack: In this attack, Eve first impersonates Bob to Alice and then imperonsates Alice to Bob, essentially conducting a MITM attack. The point of this attack, from Eve's perspective, is to determine the content of the message by measuring what she receives from Alice. Again, as above, Eve must guess the keys K_1, K_2.

5) 1-Qubit Modification Attack: In this attack, Eve performs a MITM attack, similar to above, but rather than measuring the message to determine its content, she modifies one of the qubits and then sends the resulting message to Bob.

III. Simulation of Protocol with and without Attacks

A. Summary

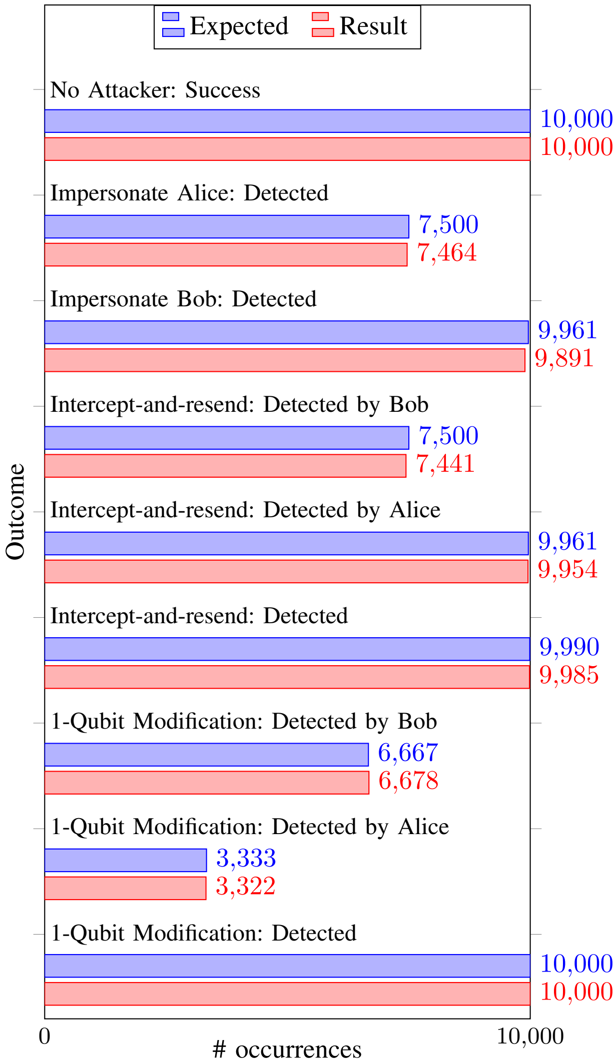
The results achieved mostly reflected the predictions of the creators of the protocol. The protocol always worked in the absence of an attacker, as expected. In the cases involving Eve either impersonating Alice or Bob, the protocol could execute successfully only in the cases where Eve guesses both keys correctly, or where both keys are wrong but in such a way that in combination they right each other.

B. Security Analysis

1) Impersonating Alice Attack: Since Eve does not know the key K_1, she must guess it. If K_1 is n bits long, then the probability of Eve guessing the correct key is $\frac{n}{2^n}$. In the event that she does guess K_1 correctly, she will successfully impersonate Alice, thus fooling Bob. In the present work, the length of the key used during simulation was $n = 12$, resulting in a probability $\frac{12}{2^{12}} = \frac{3}{1024}$ that Eve correctly guessed the key.

2) Impersonating Bob Attack: Since Eve does not know the key

Fig. 1. Expected vs. simulated results after 10,000 simulations with $n = 16$



3) *Intercept and Resend Attack:*

4) *1-Qubit Modification Attack:*

IV. Evaluation

The measured results do support the claims of the randomized-ASQDC protocol. The implementation tested in the current work behaved as expected in the face of five attacker cases: no attacker, impersonation of Alice, impersonation of Bob, intercept-and-resend attack, and 1-qubit modification attack.

V. Conclusion

In summary, the current work tested a Matlab with QIT implementation of the randomization-based ASQDC protocol in five sets of circumstances: no attacker, impersonation of Alice, impersonation of Bob, intercept-and-resend attack, and 1-qubit modification attack. The measured results were then compared against the theoretical values, supporting the claims of the ASQDC protocol authors.

This is relevant to quantum communications research as such a protocol may be more feasible than fully-quantum protocols in a variety of circumstances in the short-to-medium term. Furthermore, it was shown that the protocol is, in fact, secure against four major categories of attack.

Future work would include testing the protocol against additional attacks, such as n-qubit modification attacks. It would also be of benefit to test the protocol on computers which are capable of handling larger values of n (longer messages) in order to demonstrate empirically that the security does in fact scale with message length as predicted by theory.

References

[1] Y.-P. Luo, T. Hwang, "Authenticated semi-quantum direct communication protocols using Bell states," Quantum Information Processing, vol. 15, no. 2, pp. 957–958, 2015.

[2] K. Yu, C.-W. Yang, C.-H. Liao, T. Hwang, "Authenticated semi-quantum key distribution protocol using Bell states," Quantum Information Processing, vol. 13, no. 6, pp. 1457–1565, 2014.

Appendix

All Matlab code for the project is attached below. The `Execute.m` script was used to generate the simulated results in Fig. 1. `numExecutions` on line 5 is the number of simulations to run for each type of attack, which can be modified from 10,000.

Alice.m

```
classdef Alice < handle
    %ALICE The sender of the message.
    %   Alice has powerful quantum capabilities and quantum memory.

    properties
        % K1:[number] - Array of 0's and 1's used to generate a Lehmer
        % permutation to shuffle S + Cb to send to Bob
        K1

        % K2:[number] - Array of 0's and 1's used to generate a Lehmer
        % permutation to resore the reflected Cb received from Bob
        K2

        % checkSequence:[number] - Array of 0's and 1's representing the
        % check state C. (Used to generate Bell-EPR pairs.)
        checkSequence

        % success:bool - Set to false when Alice sends a message, and set
        % to true again when Alice receives the correct reflected check
        % state from Bob
        success
    end

    methods
        function obj = Alice(K1, K2)
            obj.K1 = K1;
            obj.K2 = K2;
        end

        function [] = sendMessage(obj, bob, m)
            %SENDMESSAGE Sends a message to Bob
            %   in  bob:Bob - Recipient
            %   in  m:[number] - message cbits

            %disp('Alice is sending a message to Bob.');
            obj.success = false;
            M = [m; utilities.hash(m)];
            S = Alice.generateBellPairs(M);
            obj.checkSequence = randi([0 1], length(M), 1);
            C = Alice.generateBellPairs(obj.checkSequence);
            Cba = Alice.separateCheckPairs(C);
            SCba = tensor(S, Cba);
            Q = utilities.LehmerShuffleK1(SCba, obj.K1);
            bob.receiveMessage(obj, Q);
        end

        function [] = receiveReflectedCheckState(obj, reflectedSCba__)
            %RECEIVEREFLECTEDCHECKSTATE Receives the reflected Cb state
            %       from Bob and reorders to bring the Bell-EPR pairs back
            %       together, verifying with the original.
            %   in  bob:Bob - Recipient of original message. Sender of the
            %       reflected check state.
            %   in  collapsedSCba_:state - The Cb portion of this state is
            %       reflected by Bob. Ca was never actually sent by Alice,
            %       and S was measured by Bob using Z-basis measurement.

            %disp('Alice is receiving the reflected check state from Bob.');
            SCba__ = utilities.LehmerShuffleK2(reflectedSCba__, obj.K2);
            SC_ = Alice.restoreCheckPairs(SCba__);
            checkSequence_ = Alice.readCheckState(SC_);
            if ~isempty(checkSequence_) && isequal(checkSequence_, obj.checkSequence)
                %disp('Alice has confirmed that Bob successfully received the message.');
                obj.success = true;
            else
                %disp('Failure: Check state reflected from Bob contained errors.');
                obj.success = false;
            end
        end
    end

    methods (Static)
        function [bellPairs] = generateBellPairs(cbits)
            %   in  cbits:[number] - input cbits for Bell-EPR pairs to
```

```
        %       generate
        %   out bellPairs:state - State containing all generated
        %       Bell-EPR pairs tensored together
        for i = 1:length(cbits)
            if cbits(i) == 0
                % Phi+
                newPair = state('Bell3');
            else
                % Psi-
                newPair = state('Bell1');
            end
            if exist('bellPairs', 'var')
                bellPairs = tensor(bellPairs, newPair);
            else
                bellPairs = newPair;
            end
        end
    end

    function [Cba] = separateCheckPairs(C)
        %   in  C:state - input qbits
        %   out Cba:state - The same qbits reordered such that the
        %       first qbit from every pair is pushed to the end. The
        %       first n/2 qbits are considered Cb and the last n/2
        %       qbits are considered Ca.

        n = C.subsystems();
        Cba = C;
        for i = 1:n/2
            % Move the first qbit from every pair to the end
            iToLast = helper.BSWAP(i, n, n);
            Cba = u_propagate(Cba, iToLast);
        end
    end

    function [SC_] = restoreCheckPairs(SCba__)
        %   in  SCba__:state - input qbits
        %   out SC_:state - The same qbits reordered such that Ca and
        %       Cb are paired back up into Bell-EPR pairs.

        % ignore S and start at Cb
        start = SCba__.subsystems() / 2 + 1;
        % number of qbits in Cba
        n = SCba__.subsystems() / 2;

        SC_ = SCba__;
        for i = start + n/2 - 1 : -1 : start
            % Move the first qbit from every pair to the end
            lastToI = helper.BSWAP(start + n - 1, i, start + n - 1);
            SC_ = u_propagate(SC_, lastToI);
        end
    end

    function [checkSequence_] = readCheckState(SC_)
        %   in  SC_:state - Contains check state C which should contain
        %       the originally generated Bell-EPR pairs.
        %   out checkSequence_:[number] - cbits obtained by performing
        %       Bell measurement on all pairs in C.

        % start with I gates for all qbits in S
        prep = helper.power(helper.I, SC_.subsystems() / 2);
        % ignore S and start at Cb
        start = SC_.subsystems() / 2 + 1;
        % number of qbits in Cba
        n = SC_.subsystems() / 2;

        % add Bell-measurement preparation gates for every pair of qbits
        for i = start : 2 : start + n - 1
            prep = tensor(prep, tensor(helper.H, helper.I) * helper.ACNOT);
        end
        preppedSC_ = u_propagate(SC_, prep);

        % measure each qbit in C
        checkSequence_ = zeros(n/2, 1);
```

```

            for i = start : 2 : start + n - 1
                [~, b1, preppedSC_] = measure(preppedSC_, i);
                b1 = b1 - 1;
                [~, b2, preppedSC_] = measure(preppedSC_, i+1);
                b2 = b2 - 1;

                if b1 == 0 && b2 == 0
                    % if the bits are the same, Alice sent a Phi+
                    % which represents 0
                    checkSequence_((i - start)/2 + 1) = 0;
                elseif b1 == 1 && b2 == 1
                    % if the bits are the different, Alice sent a Psi-
                    % which represents 1
                    checkSequence_((i - start)/2 + 1) = 1;
                else
                    %disp('Error: unexpected Bell-measurement reading');
                    checkSequence_ = [];
                    return;
                end
            end
        end
    end
end
```

Bob.m

```
classdef Bob < handle
    %BOB The receiver of the message.
    %   Bob has only classic capabilities and Z-basis measurement.

    properties
        % K1:[number] - Array of 0's and 1's used to generate a Lehmer
        % permutation to restore S + Cb received from to Bob
        K1

        % K2:[number] - Array of 0's and 1's used to generate a Lehmer
        % permutation to shuffle the reflected Cb sent to Bob
        K2

        % receivedMessage:[number] - Array of 0's and 1's representing the
        % message sent from Alice. If the calculated hash is invalid, the
        % protocol is aborted and this value is cleared.
        receivedMessage

        % hashVerified:[boolean] - Set to true if the hash was valid on the
        % last message, false otherwise
        hashVerified
    end

    methods
        function obj = Bob(K1, K2)
            obj.K1 = K1;
            obj.K2 = K2;
        end

        function [] = receiveMessage(obj, alice, Q_)
            %RECEIVEMESSAGE Receives a message from Alice
            %   in  alice:Alice - Sender
            %   in  Q:state - message as a quantum state containing the
            %       message component S which is shuffled with the 2nd part
            %       of the check component Cb, followed by the first part
            %       of the check component Ca. Ca is technically not sent
            %       by Alice and should not be touched by Bob in this
            %       simulation.

            %disp('Bob is receiving a message.');
            SCba_ = utilities.LehmerShuffleK1(Q_, obj.K1);
            [M_, collapsedSCba_] = Bob.readMessage(SCba_);
            [m_, obj.hashVerified] = Bob.verifyHash(M_);
            obj.receivedMessage = m_;
            if obj.hashVerified
                %disp('Bob successfully received the message.');
                %disp('Bob is reflecting the check state back to Alice.');
```

```
                shuffledSCba_ = utilities.LehmerShuffleK2(collapsedSCba_, obj.K2);
                alice.receiveReflectedCheckState(shuffledSCba_);
            else
                %disp('Failure: Incorrect hash on message received by Bob.');
            end
        end
    end

    methods (Static)
        function [M_, collapsedSCba_] = readMessage(SCba_)
            n = SCba_.subsystems();
            M_ = zeros(n/4, 1);
            collapsedSCba_ = SCba_;
            for i = 1:2:n/2
                [~, b1, collapsedSCba_] = measure(collapsedSCba_, i);
                b1 = b1 - 1;
                [~, b2, collapsedSCba_] = measure(collapsedSCba_, i+1);
                b2 = b2 - 1;

                if b1 == b2
                    % if the bits are the same, Alice sent a Phi+
                    % which represents 0
                    M_((i-1)/2 + 1) = 0;
                else
                    % if the bits are the different, Alice sent a Psi-
                    % which represents 1
                    M_((i-1)/2 + 1) = 1;
                end
            end
        end

        function [m_, success] = verifyHash(M_)
            m_ = M_(1 : length(M_)/2);
            h_ = M_(length(M_)/2 + 1 : length(M_));
            success = isequal(h_, utilities.hash(m_));
        end
    end
end
```

Eve.m

```
classdef Eve
    %EVE Summary of this class goes here
    %   Detailed explanation goes here

    properties
        n
        % NOTE: n bit key according to protocol description, but we'll just store
        % a Lehmer code because it makes way more sense
        eK1
        % NOTE: n/2 bit key according to protocol description, but we'll just store
        % a Lehmer code because it makes way more sense
        eK2
        eAlice
        eBob
    end

    methods
        function obj = Eve(n)
            obj.n = n;
            % NOTE: n bit key according to protocol description, but we'll just store
            % a Lehmer code because it makes way more sense
            obj.eK1 = utilities.createLehmerCode(n*3/4);
            % NOTE: n/2 bit key according to protocol description, but we'll just store
            % a Lehmer code because it makes way more sense
            obj.eK2 = utilities.createLehmerCode(n*1/4);
            obj.eAlice = Alice(obj.eK1, obj.eK2);
            obj.eBob = Bob(obj.eK1, obj.eK2);
        end

        function [] = impersonateAlice(obj, bob, m)
            %IMPERSONATION Sends a message to Bob using random keys
            %   in  bob:Bob - Recipient
            %   in  m:[number] - message cbits
```

```
            obj.eAlice.sendMessage(bob, m);
        end

        function [] = impersonateBob(obj, alice, m)
            %IMPERSONATION Sends a message to Bob using random keys
            %   in  bob:Bob - Recipient
            %   in  m:[number] - message cbits

            %disp('Alice is sending a message to "Bob" (Eve).');
            alice.success = false;
            M = [m; utilities.hash(m)];
            S = Alice.generateBellPairs(M);
            alice.checkSequence = randi([0 1], length(M), 1);
            C = Alice.generateBellPairs(alice.checkSequence);
            Cba = Alice.separateCheckPairs(C);
            SCba = tensor(S, Cba);
            Q = utilities.LehmerShuffleK1(SCba, alice.K1);

            Q_ = Q;

            %disp('"Bob" (Eve) is receiving a message.');
            SCba_ = utilities.LehmerShuffleK1(Q_, obj.eK1);
            [M_, collapsedSCba_] = Bob.readMessage(SCba_);
            [m_, obj.eBob.hashVerified] = Bob.verifyHash(M_);
            obj.eBob.receivedMessage = m_;
            if obj.eBob.hashVerified
                %disp('Eve successfully received the message.');
                %disp('Eve is reflecting the check state back to Alice.');
            else
                %disp('Failure: Incorrect hash on message received by Eve.');
                %disp('Eve is reflecting the check state back to Alice anyway.');
            end
            shuffledSCba_ = utilities.LehmerShuffleK2(collapsedSCba_, obj.eK2);
            alice.receiveReflectedCheckState(shuffledSCba_);
        end

        function [] = interceptResend(obj, alice, bob, m)
            % INTERCEPTRESEND Receives a message from Alice, measures it,
            % and resends it to Bob
            %   in  alice:Alice - Sender
            %   in  Q:state - message as a quantum state containing the
            %       message component S which is shuffled with the 2nd part
            %       of the check component Cb, followed by the first part
            %       of the check component Ca. Ca is technically not sent
            %       by Alice and should not be touched by Bob in this
            %       simulation.
            %   in  bob:Bob - Recipient
            %   in  m:[number] - message cbits
            alice.sendMessage(obj.eBob, m);
            obj.eAlice.sendMessage(bob, obj.eBob.receivedMessage);
        end

        function [] = modification(obj, alice, bob, m)
            % MODIFICATION Receives a message from Alice, changes it,
            % and resends it to Bob
            %   in  alice:Alice - Sender
            %   in  Q:state - message as a quantum state containing the
            %       message component S which is shuffled with the 2nd part
            %       of the check component Cb, followed by the first part
            %       of the check component Ca. Ca is technically not sent
            %       by Alice and should not be touched by Bob in this
            %       simulation.
            %   in  bob:Bob - Recipient
            %   in  m:[number] - message cbits

            %disp('Alice is sending a message to Bob.');
            alice.success = false;
            M = [m; utilities.hash(m)];
            S = Alice.generateBellPairs(M);
            alice.checkSequence = randi([0 1], length(M), 1);
            C = Alice.generateBellPairs(alice.checkSequence);
            Cba = Alice.separateCheckPairs(C);
            SCba = tensor(S, Cba);
            Q = utilities.LehmerShuffleK1(SCba, alice.K1);
```

```
            % Eve modifies a single qubit
            A_ind = randi(obj.n*3/4);
            if A_ind == 1
                A = tensor(helper.X, helper.power(helper.I, obj.n-A_ind));
            else
                A = tensor(helper.power(helper.I, A_ind-1), helper.X, helper.power(helper.I, obj.n-A_ind));
            end
            Aq = u_propagate(Q,A);
            bob.receiveMessage(alice, Aq);
        end
    end
end
```

Execute.m

```
% This is the main execution script, which executes each of the attacks
% on a loop and counts the outcomes.

n = 16;
numExecutions = 10000;

fprintf('Executing randomization-based protocol, no attacker...\n');
successCount = 0;
for i = 1:numExecutions
    [m, alice, bob, ~] = prepare(n);
    alice.sendMessage(bob, m);

    if isequal(bob.receivedMessage, m) && bob.hashVerified && alice.success
        successCount = successCount + 1;
    end
end
fprintf('Number of simulations: %+5s\n', sprintf('%d', numExecutions));
fprintf('Protocol succeeded:    %+5s\n\n', sprintf('%d', successCount));

fprintf('Executing impersonation of Alice attack on randomization-based protocol...\n');
hashVerifiedCount = 0;
for i = 1:numExecutions
    [m, alice, bob, eve] = prepare(n);
    eve.impersonateAlice(bob, m);

    if bob.hashVerified
        hashVerifiedCount = hashVerifiedCount + 1;
    end
end
fprintf('Number of simulations:   %+5s\n', sprintf('%d', numExecutions));
fprintf('Eve was detected by Bob: %+5s\n\n', sprintf('%d', numExecutions - hashVerifiedCount));

fprintf('Executing impersonation of Bob attack on randomization-based protocol...\n');
checkVerifiedCount = 0;
for i = 1:numExecutions
    [m, alice, bob, eve] = prepare(n);
    eve.impersonateBob(alice, m);

    if alice.success
        checkVerifiedCount = checkVerifiedCount + 1;
    end
end
fprintf('Number of simulations:     %+5s\n', sprintf('%d', numExecutions));
fprintf('Eve was detected by Alice: %+5s\n\n', sprintf('%d', numExecutions - checkVerifiedCount));

fprintf('Executing intercept and resend attack randomization-based protocol...\n');
hashVerifiedCount = 0;
checkVerifiedCount = 0;
undetectedCount = 0;
for i = 1:numExecutions
    [m, alice, bob, eve] = prepare(n);
    eve.interceptResend(alice, bob, m);

    if bob.hashVerified
        hashVerifiedCount = hashVerifiedCount + 1;
        if alice.success
            undetectedCount = undetectedCount + 1;
        end
    end
```

```
    if alice.success
        checkVerifiedCount = checkVerifiedCount + 1;
    end
end
fprintf('Number of simulations:             %+5s\n', sprintf('%d', numExecutions));
fprintf('Eve was detected by Bob:           %+5s\n', sprintf('%d', numExecutions - hashVerifiedCount));
fprintf('Eve was detected by Alice:         %+5s\n', sprintf('%d', numExecutions - checkVerifiedCount));
fprintf('Eve was detected by at least one:  %+5s\n\n', sprintf('%d', numExecutions - undetectedCount));

fprintf('Executing 1-qbit modification attack on randomization-based protocol...\n');
hashVerifiedCount = 0;
checkVerifiedCount = 0;
bothVerifiedCount = 0;
for i = 1:numExecutions
    [m, alice, bob, eve] = prepare(n);
    eve.modification(alice, bob, m);

    if bob.hashVerified
        hashVerifiedCount = hashVerifiedCount + 1;
        if alice.success
            bothVerifiedCount = bothVerifiedCount + 1;
        end
    end
    if alice.success
        checkVerifiedCount = checkVerifiedCount + 1;
    end
end
fprintf('Number of simulations:     %+5s\n', sprintf('%d', numExecutions));
fprintf('Eve was detected by Bob:   %+5s\n', sprintf('%d', numExecutions - hashVerifiedCount));
fprintf('Eve was detected by Alice: %+5s\n', sprintf('%d', hashVerifiedCount - checkVerifiedCount));
fprintf('Eve was detected:          %+5s\n', sprintf('%d', numExecutions - bothVerifiedCount));


function [m, alice, bob, eve] = prepare(n)
    m = randi([0 1], n/8, 1);
    % NOTE: this should be an n bit key according to protocol description,
    % but we'll just store a Lehmer code because it makes way more sense
    K1Encode = utilities.createLehmerCode(n*3/4);
    K1Decode = utilities.invertLehmerCode(K1Encode);
    % NOTE: this should be an n/2 bit key according to protocol
    % description, but we'll just store a Lehmer code because it makes way
    % more sense
    K2Encode = utilities.createLehmerCode(n*1/4);
    K2Decode = utilities.invertLehmerCode(K2Encode);

    alice = Alice(K1Encode, K2Decode);
    bob = Bob(K1Decode, K2Encode);
    eve = Eve(n);
end
```

helper.m

```
% Quantum Programming Helper
% Author: Michel Barbeau
% Version: February 8, 2016
% Dependency: Quantum Information Toolkit
classdef helper < handle
    % static properties
    properties (Constant)
        % Gates
        ACNOT = gate.mod_add(2, 2)
        BCNOT = helper.ACNOT * helper.SWAP * helper.ACNOT
        H = gate.qft(2)
        I = gate.id(2)
        SWAP = gate.swap(2,2)
        X = gate.mod_inc(-1, 2)
        Y = 1 / i * helper.Z * helper.X
        Z = helper.H * gate.mod_inc(1, 2) * helper.H'
        % Bell-EPR production
        E = helper.ACNOT * tensor(helper.H, helper.I);
        % Bell-EPR measurement preperation
        prep = tensor(helper.H, helper.I) * helper.ACNOT;
    end
    methods (Static)
```

```
        function [ P ] = power(G, n)
            % return the nth tensor power of gate G (n?0)
            % WARNING: power(G,0) does not work with tensor function
            if (n == 0)
                P = lmap(1, {[ 1 1 ]});
            else
                P = G;
                for i = 2 : n
                    P = tensor(P, G);
                end
            end
        end

        function [ G ] = R(k, n)
            % return a n by n gate swapping qubits k and k+1
            if (k <= 0 || k >= n)
                error('In function R: must have 0<k<n');
            end
            if (k + 1 < n)
                G = tensor(helper.SWAP, helper.power(helper.I, n - (k + 1)));
            else
                G = helper.SWAP;
            end
            if (k > 1)
                G = tensor(helper.power(helper.I, k - 1), G);
            end
        end

        function [ G ] = BSWAP(k, l, n)
            % if k<l
            % ret. a n*n gate swapping qubit k and qubits k+1 to l
            % if k>l
            % ret. a n*n gate swapping qubits l to k-1 and qubit k
            % else
            % ret. a n*n I gate
            if (k <= 0 || k > n)
                error('In function BSWAP: must have 0<k?n');
            end
            if (l <= 0 || l > n)
                error('In function BSWAP: must have 0<l?n');
            end
            G = helper.power(helper.I, n);
            if (k < l)
                for m = k : l - 1
                    G = helper.R(m, n) * G;
                end
            elseif (k > l)
                for m = k - 1 : -1 : l
                    G = helper.R(m, n) * G;
                end
            end
        end

        function [ S ] = ebit(in)
            % return a Bell-EPR state
            % in=input state, e.g., |00>
            S = u_propagate(in, helper.E);
        end
    end
end
```

utilities.m

```
classdef utilities
    methods (Static)
        function [code] = createLehmerCode(len)
            %def createLehmerCode(len):
            %   result = []
            %   for i in range(len, 0, -1):
            %       result.append(random.randint(0,i))
            %   return result
            code = [];
            for i = len : -1 : 1
                code = [code randi(i)];
```

```
            end
        end

        function [e] = remove(elems, i)
            a = elems(1:(i-1));
            b = elems((i+1):length(elems));
            e = [a b];

        end

        function [perm] = permuteFromCode(elems, code)
            %def codeToPermutation(elems, code):
            %  def f(i):
            %      e=elems.pop(i)
            %      return e
            %  return list(map(f, code))

            %x = @(f) remove(
            %perm = []
            %perm = maprec(code, pop(elems,i));

            e = elems(1:length(elems));
            perm = [];
            for i = 1 : length(code)
                perm = [perm e(code(i))];
                e = utilities.remove(e,code(i));
            end
        end

        function [perm] = codeToPermutation(code)
            perm = code;
            n = length(perm);
            for i = n:-1:1
                for j = i+1:n
                    if perm(j) >= perm(i)
                        perm(j) = perm(j) + 1;
                    end
                end
            end
        end

        function [inv] = invertPermutation(perm)
            n = length(perm);
            inv = zeros(n, 1);
            for i = 1:n
                x = perm(i);
                inv(x) = i;
            end
        end

        function [code] = permutationToCode(perm)
            code = perm;
            n = length(code);
            for i = 1:n
                for j = i+1:n
                    if code(j) > code(i)
                        code(j) = code(j) - 1;
                    end
                end
            end
        end

        function [invCode] = invertLehmerCode(code)
            perm = utilities.codeToPermutation(code);
            invPerm = utilities.invertPermutation(perm);
            invCode = utilities.permutationToCode(invPerm);
        end

        function [num] = bitArrayToNumber(bits)
            n = length(bits);
            num = 0;
            for exp = 0:n-1
                i = n - exp;
                if bits(i) == 1
```

```
                    num = num + 2^exp;
                end
            end
        end

        function [code] = integerToCode(K, n)
            %def integerToCode(K, n):
            %   if (n<=1):
            %       return [0]
            %   multiplier = factorial(n-1)
            %   digit = floor(K/multiplier)
            %   r = [digit]
            %   r.extend(integerToCode(K%multiplier, n-1))
            %   return r

            if n <= 1
                code = [1];
            else
               multiplier = factorial(n-1);
               digit = floor(K/multiplier)+1;
               if (digit > n)
                   error('K >= n!');
               end
               code = [digit utilities.integerToCode(rem(K,multiplier), n-1)];
            end
        end

        function [h] = hash(m)
            %%%   h:string
            %%%   m:string
            %import java.security.*;
            %import java.math.*;
            %%% instantiate Java MessageDigest using MD5
            %md = MessageDigest.getInstance('MD5');
            %%% convert m to ASCII numerical rep in base-64 radix
            %h_array = md.digest(double(m));
            %%% convert int8 array into Java BigInteger
            %bi = BigInteger(1, h_array);
            %%% convert hash into string format
            %hStr = char(bi.toString(2));
            %%% convert string to bit array
            %h = zeros(128, 1);
            %for i = 1:length(hStr)
            %    h(length(h) + 1 - i) = str2num(hStr(length(hStr) + 1 - i));
            %end


            % Ignore the good hash function above and use this poor one
            % because we can only afford a few bits. This not very secure
            % but guaranteed to be collision free. In practice, we would
            % use the above function, but we cannot simulate the protocol
            % with that many qubits on an ordinary computer (not enough
            % RAM).
            h = m;
            for i = 1:length(h)
                if h(i) == 0
                    h(i) = 1;
                else
                    h(i) = 0;
                end
            end
        end

        function [outState] = LehmerShuffle(inState, code, first, last)
            n = inState.subsystems();
            if length(code) ~= last - first + 1
                error('Key length does not match number of elements');
            end

            outState = inState;
            for i = 1 : last - first + 1
                index = first + code(i) - 1;
                iToLast = helper.BSWAP(index, last, n);
                outState = u_propagate(outState, iToLast);
```

```
            end
        end

        function [Q] = LehmerShuffleK1(SCba, K1)
            %   in  SCba:state - S is the tensored Bell-EPR pairs
            %       representing the message and message hash components.
            %       Cb is the sequence of 1st qbits from each of the check
            %       state Bell-EPR pairs, and Ca is the 2nd qbit from each.
            %   in  K1:[number] - cbit Key used for reordering S+Cb according to
            %       the Lehmer code algorithm.
            %   out Q:state - Tensored and reordered output state. Ca stays
            %       the same but S and Cb are shuffled together according
            %       to K1.

            n = SCba.subsystems();
            Q = utilities.LehmerShuffle(SCba, K1, 1, n*3/4);
        end

        function [shuffledSCba_] = LehmerShuffleK2(SCba_, K2)
            %   in  SCba_:state - State as received from Alice (ignore Ca).
            %       We will shuffle the S+Cb component back to its original
            %       order.
            %   in  K2:[number] - cbit Key used for reordering Cb according to
            %       the Lehmer code algorithm.
            %   out shuffledSCba_:state - Reordered output state. Ca and S
            %       stay the same but Cb is shuffled according to K2.

            n = SCba_.subsystems();
            shuffledSCba_ = utilities.LehmerShuffle(SCba_, K2, n*1/2 + 1, n*3/4);
        end

        function [str] = bitstring(bitArray)
            str = "";
            for i = 1:length(bitArray)
                if bitArray(i) == 0
                    str = str + "0";
                else
                    str = str + "1";
                end
            end
        end
    end
end
```