Documentation to privacy-amplification – INTERNAL DRAFT –

Robert Riemann

January 13, 2014

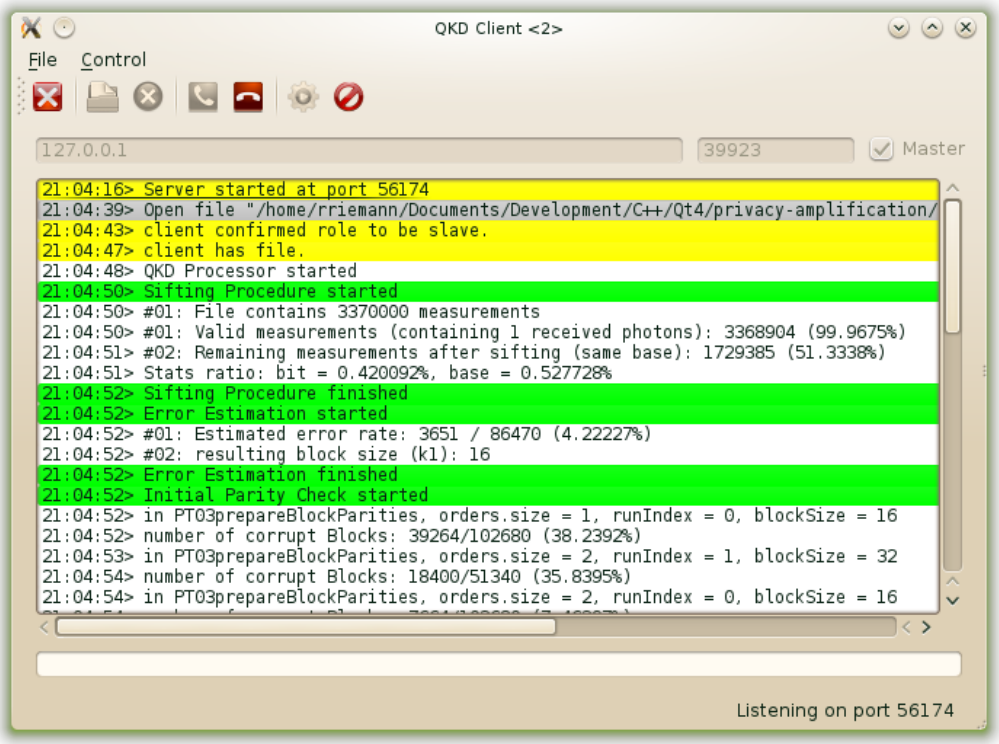

Figure 1: Screenshot of the QKD Client in Alice-mode

Note on Qt (Framework)

This software uses the cross-platform C++ framework Qt (pronounced like "cute", LPGL licensed). Hence there are some C++ language extensions.

- Macros starting with `Q_`
- additional keywords like `foreach` and `emit`

`emit` is part of the Signals/Slots mechanism[1] and is used to trigger a so called Signal. Other objects, which are listening to that *Signal* might react by processing their connected *Slots*. The code execution doesn't take place immediately, but anytime later and is triggered by the global event loop.

This means that emitting a signal doesn't cause any delay, even if connected slots rely on network.

Qt comes with a comprehensive documentation which is also available online at:
`http://qt-project.org/doc/qt-4.8/`

Detailed Description

The most important stuff is in `QKDProcessor::incomingData(quint8 type, QVariant data)`, which is defined in `qkdprocessor.cpp`.

On loading this application, one `QKDProcessor` Object is instanciated. The `Mainwindow` Object starts the protocol by calling `QKDProcessor::start(bool isMaster)` after is has been made sure, that a file was loaded and the measurements were assigned by `QKDProcessor::setMeasurements(Measurements measurements)`.

Measurements are usually allocated onto the heap (using `new`) and consists of 3 boolean properties:

valid
: a flag which marks this measurement as valid. This is used on Bob's side to mark measurements with more than two registrated photons as invalid.

base
: a flag to save the state accordingly to the chosen base out of 2.

bit
: a flag to save the bit which was transfered. These bits constitutes later on our cryptographic key.

The list of pointers to all measurements is managed by a `QList` objects, which is itself also allocated onto the heap to make it easy to pass these measurements between a file and our `QKDProcessor` object.

Listing 1: extract of measurement.h

```
1  struct Measurement
2  {
3      Measurement(bool base, bool bit, bool valid = 1) : base(base), bit(bit), valid(valid) {}
4      explicit Measurement() : base(0), bit(0), valid(1) {}
5      Measurement &operator=(const Measurement &other)
6      {base = other.base; bit = other.bit; valid = other.valid; return *this;}
7
8      bool base;
9      bool bit;
10     bool valid;
11 };
12 typedef QList<Measurement*> Measurements;
```

After the processing has started, the `QKDProcessor::incomingData(quint8 type, QVariant data)` is called alternately by Alice, also known as `isMaster` (`bool`), and Bob, respectively `!isMaster`.

[1] `http://qt-project.org/doc/qt-4.8/signalsandslots.html`

Important Variables

To keep the variable declaration right next to the code, most variables used to implement the protocol are not declared as member variables of `QKDProcessor`, but as `static` local variables right on top of `incomingData` method.

`qreal` **error**
: Error probability. The error probabilities estimated by comparing 0,05 % of all bits. These bits were deleted immediately afterward.

`quint16` **k1**
: Initial block size calculated by `calculateInitialBlockSize(qreal errorProbability)`.

`quint8` **runCount**
: This constant integer setups the limit of individual reorderings of all data. Each consecutive reordering doubles the block size starting with `k1`.

`quint8` **runIndex**
: Run index counts towards `runCount`.

`quint16` **blockSize**
: The actual block size can be determined using `runCount` and `k1`.

`Index` **blockCount** The total count of blocks depends on the total count of measurements as well as the current `blockSize`. To overcome issues with incomplete blocks, the overlap gets deleted initially.

Program Run

1. measurement data get loaded using `setMeasurements` method
2. protocol gets started using `start` method
3. Alice asks Bob to send an `IndexList` of received photons (`PT01sendReceivedList`).
4. Bob answers by sending a list of indexes with his valid measurement including with his chosen base.
5. Alice calculated the remaining measurements taking only measurements with same base into account.
6. Alice sends a list of indexes of his remaining measurement to Bob. (`PT01sendRemainingList`)
7. Bob receives this list, reports his to Alice and both do the sifting using `siftMeasurements`.
8. Alice requires Bob to send back a specific count of bits (taken from the last measurements) to estimate the error probability (`PT02errorEstimationSendSample`).
9. Bob responds by sending this amount of bits and deleting the appropriate measurements immediately.
10. Alice compares the bits to calculate the error probability (`error`) and reports the absolute count of errors back to Bob who calculates the same error on his own (`PT02errorEstimationReport`).
11. Both parties calculate the inital block size `k1` which depends `error`. Afterwards the parities are calculated at the same time. The initial bit order is set up to be chronological (*about to change soon!*).

12. Alice sends Bob the order to report his blocks with wrong parities (corrupt block) given Alices parities (`PT04reportBlockParities`).

    As the block size is known to both (first `blockSize`, `binaryBlockSize` while doing BINARY), it is sufficient to only refer to the index of the first measurement of the corrupt block given the ordering related to the current `runIndex`/`blockSize`.

13. Bob does so and Alice receives the (maybe empty) list of corrupt blocks. Alice has to decide what has to be done next:

    a) start or continue BINARY

    b) create a new random ordering of all measurements and send it to Bob to enter the next level (`runIndex`)

    c) go back to first level (`runIndex`) to correct new appearing corrupt blocks

    d) enter the next higher level which has already been taken and therefor it is not necessary to calculate and send a new order again

    e) finish the error correction as the last level (`runCount`) has been reached

    Please read the code of `PT04reportBlockParities` for Alice (`if(isMaster)...`) to understand this decision in detail.

14. For testing purposes there is a complete key exchange afterwards to calculate the remaining error rate.

Source Code

Listing 2: qkdprocessor.cpp

```
#include "qkdprocessor.h"

#include <qmath.h>
#include <qdebug.h>
#include <QTime>
#include <QRgb>
#include <QFile>
#include <limits>
#include <algorithm>
using std::max;
using std::random_shuffle;

const int QKDProcessor::idIndexList = qRegisterMetaType<IndexList>();

QKDProcessor::QKDProcessor(QObject *parent) :
    QObject(parent), measurements(0), isMaster(false)
{
    // make sure tu initialize the pseudo RNG for std::random_shuffle
#ifdef QT_NO_DEBUG
    qsrand(QTime::currentTime().msec()); srand(QTime::currentTime().msec()+1);
#else
    qsrand(0); srand(0); // make the class deterministic for debugging purposes
#endif

    qRegisterMetaTypeStreamOperators<IndexList>();

    qRegisterMetaType<IndexBoolPair>();
    qRegisterMetaTypeStreamOperators<IndexBoolPair>();

    qRegisterMetaType<IndexBoolPairList>();
    qRegisterMetaTypeStreamOperators<IndexBoolPairList>();

    qRegisterMetaType<BoolList>();
    qRegisterMetaTypeStreamOperators<BoolList>();
}

void QKDProcessor::clearMeasurements()
{
    if(measurements) {
        qDeleteAll(*measurements);
        delete measurements;
    }
}

QByteArray QKDProcessor::privacyAmplification(const Measurements measurements, const qreal ratio)
{
    Q_ASSERT(ratio > 0);
    /* - count of bits to represent integer: N=ceil(log2(int))
     * - count of bits to represent squared integer: 2N
     * - must have double size of bufferTypeSmall to prevent integer overflow
     * - in any case only half of the bits of squared integer is used, so we can
     * accept integer overflow and use the same bit size for bufferType, too.
     * - bigger types are better, because precision increasing in:
     * floor(bitLimitSmall*ratio). step size: 100/64 => 1.56%
     */
    typedef quint64 bufferType;
    typedef quint64 bufferTypeSmall;

    static const quint8 bitLimitSmall = sizeof(bufferTypeSmall)*8;

    bufferTypeSmall bufferBits = 0;
    bufferTypeSmall bufferBase = 1; // bufferBase must be odd (-> hash functions)

    QByteArray finalKey;
    // only cstrings and char can be added to QByteArrays: we choose char
    typedef char finalKeyBufferType;

    finalKeyBufferType buffer = 0;
    const quint8 bufferSize = sizeof(finalKeyBufferType)*8;
```

```
70      const quint8 bitCount = qFloor(bitLimitSmall*ratio);
71
72      quint8 pos = 0;
73      quint8 bufferPos = 0;
74      const bufferType bufferType_1 = Q_UINT64_C(1);
75      foreach(Measurement *measurement, measurements) {
76          bufferBase |= (bufferTypeSmall)measurement->base << pos;
77          bufferBits |= (bufferTypeSmall)measurement->bit << pos;
78          pos++;
79          if(pos == bitLimitSmall) {
80              pos = 0;
81              bufferType temp = (bufferType)bufferBase*(bufferType)bufferBits;
82              for(quint8 bitPos = 0; bitPos < bitCount; bitPos++) {
83                  // http://stackoverflow.com/a/2249738/1407622
84                  buffer |= ((temp & ( bufferType_1 << bitPos )) >> bitPos) << bufferPos;
85                  bufferPos++;
86                  if(bufferPos == bufferSize) {
87                      bufferPos = 0;
88                      finalKey.append(buffer);
89                      buffer = 0;
90                  }
91              }
92              bufferBase = 1;
93              bufferBits = 0;
94          }
95      }
96      // there might be fewer bits in finalKey than ratio*measurements->size()
97      // this is due remaining (unused) bits in buffers
98      return finalKey;
99  }
100
101 void QKDProcessor::setMeasurements(Measurements *measurements)
102 {
103     clearMeasurements();
104     this->measurements = measurements;
105 }
106
107 void QKDProcessor::setMaster(bool isMaster)
108 {
109     this->isMaster = isMaster;
110 }
111
112 QKDProcessor::~QKDProcessor()
113 {
114     clearMeasurements();
115 }
116
117 quint16 QKDProcessor::calculateInitialBlockSize(qreal errorProbability)
118 {
119     // values follow the proposal of Brassard and Savail(1994), but are
120     // aligned to the power of 2 which is necessary for the binary scheme
121     if(errorProbability < 0.02) {
122         return 64;
123     } else if(errorProbability < 0.05) {
124         return 16;
125     } else if(errorProbability < 0.1) {
126         return 8;
127     } else if(errorProbability < 0.15) {
128         return 4;
129     } else {
130         emit logMessage(QString("EE: errorEstimation exceeded secure limit!"));
131         return 4;
132     }
133 }
134
135 bool QKDProcessor::calculateParity(const Measurements measurements,
136                                    const Index index, const quint16 &size) const
137 {
138     Q_ASSERT(size > 0);
139     const Index end = index + size;
140     Q_ASSERT((SIndex)end <= measurements.size());
141     bool parity = false;
142     for(Index i = index; i < end; i++) {
143         parity = parity ^ measurements.at(i)->bit;
```

```
144     }
145     return parity;
146 }
147 
148 IndexList QKDProcessor::getOrderedList(Index range)
149 {
150     IndexList orderedList;
151     for(Index i = 0; i < range; i++)
152         orderedList.append(i);
153     return orderedList;
154 }
155 
156 
157 IndexList QKDProcessor::getRandomList(Index range)
158 {
159     IndexList list = getOrderedList(range);
160     // http://www.cplusplus.com/reference/algorithm/random_shuffle/
161     random_shuffle(list.begin(), list.end());
162     return list;
163 }
164 
165 Measurements QKDProcessor::reorderMeasurements(const Measurements measurements, const IndexList order)
166 {
167     Measurements list;
168     foreach(Index index, order) {
169         list.append(measurements.at(index));
170     }
171 
172     return list;
173 }
174 
175 void QKDProcessor::incomingData(quint8 type, QVariant data)
176 {
177 
178     switch((PackageType)type) {
179     // Sifting Procedure
180     case PT01sendReceivedList: {
181         emit logMessage(QString("Sifting Procedure started"), Qt::green);
182         emit logMessage(QString("#01: File contains %1 measurements").
183                         arg(measurements->size()));
184         if(isMaster) {
185             list = data.value<IndexBoolPairList>();
186         } else {
187             list.clear();
188             Q_ASSERT((SIndex)std::numeric_limits<Index>::max() >= measurements->size());
189             for(int index = 0; index < measurements->size(); index++) {
190                 if(measurements->at(index)->valid)
191                     list.append(IndexBoolPair(index,measurements->at(index)->base));
192             }
193             emit sendData(PT01sendReceivedList, QVariant::fromValue(list));
194         }
195         emit logMessage(QString("#01: Valid measurements (containing 1 received photons): %1 (%2%)").
196                         arg(list.size()).arg((double)list.size()*100/measurements->size()));
197 
198         if(isMaster) {
199             IndexBoolPair pair;
200             remainingList.clear();
201             BoolList basesList;
202             foreach(pair, list) {
203                 const bool &ownBase = measurements->at(pair.first)->base;
204                 basesList.append(ownBase);
205                 if(ownBase == pair.second)
206                     remainingList.append(pair.first);
207             }
208             emit sendData(PT01sendRemainingList,
209                           QVariant::fromValue(basesList));
210         }
211 
212         reorderedMeasurements.clear();
213 
214         return;
215     }
216     case PT01sendRemainingList: {
217         if(isMaster) {
```

```
218         } else {
219             BoolList basesList = data.value<BoolList>();
220             Q_ASSERT(list.size() == basesList.size());
221             remainingList.clear();
222             for(SIndex index = 0; index < list.size(); index++) {
223                 const IndexBoolPair &pair = list.at(index);
224                 if(basesList.at(index) == pair.second)
225                     remainingList.append(pair.first);
226             }
227             emit sendData(PT01sendRemainingList);
228         }
229         emit logMessage(QString("#02: Remaining measurements after sifting (same base): %1 (%2%)").
230                         arg(remainingList.size()).
231                         arg((double)remainingList.size()*100/list.size()));
232         {
233             Measurements siftedMeasurements;
234             foreach(Index index, remainingList) {
235                 siftedMeasurements.append(measurements->at(index));
236             }
237             reorderedMeasurements.append(siftedMeasurements);
238 
239         }
240 
241         {
242             int bit = 0;
243             int base = 0;
244             Measurements measurements = reorderedMeasurements.first();
245             foreach(Measurement *measurement, measurements) {
246                 bit += measurement->bit;
247                 base += measurement->base;
248             }
249 
250             emit logMessage(QString("Stats ratio: bit = %1%, base = %2%").
251                             arg(100.0*bit/measurements.size()).
252                             arg(100.0*base/measurements.size()));
253         }
254 
255         list.clear();
256         remainingList.clear();
257         emit logMessage(QString("Sifting Procedure finished"), Qt::green);
258 
259         if(isMaster) {
260             IndexList order = getRandomList(reorderedMeasurements.first().size());
261             reorderedMeasurements.replace(0,reorderMeasurements(reorderedMeasurements.last(),
262                                                                 order));
263             emit sendData(PT02errorEstimationSendSample,
264                           QVariant::fromValue<IndexList>(order));
265         }
266         return;
267     }
268     // Error Estimation
269     case PT02errorEstimationSendSample: {
270         emit logMessage(QString("Error Estimation started"), Qt::green);
271         if(!isMaster) {
272             IndexList order = data.value<IndexList>();
273             reorderedMeasurements.replace(0,reorderMeasurements(reorderedMeasurements.last(),
274                                                                 order));
275         }
276         Q_ASSERT(reorderedMeasurements.size() == 1);
277         Q_ASSERT(reorderedMeasurements.first().size() > 1);
278         errorEstimationSampleSize = qCeil(reorderedMeasurements.first().size()*
279                                           errorEstimationSampleRatio);
280 
281         boolList.clear();
282         for(Index i = 0; i < errorEstimationSampleSize; i++) {
283             /*
284              * takeLast() removes elements from the list
285              * new orderings will take the size of the last ordering into account
286              */
287             boolList.append(reorderedMeasurements.first().takeLast()->bit);
288         }
289 
290         if(!isMaster) {
291             emit sendData(PT02errorEstimationSendSample,
```

```
                        QVariant::fromValue(boolList));
        } else {
            BoolList compareBoolList = data.value<BoolList>();
            int size = boolList.size();
            Q_ASSERT(size == compareBoolList.size());
            errorCounter = 0;
            for(int i = 0; i < size; i++) {
                if(boolList.at(i) != compareBoolList.at(i))
                    errorCounter++;
            }
            emit sendData(PT02errorEstimationReport,
                        QVariant::fromValue<Index>(errorCounter));
        }
        return;
    }
    case PT02errorEstimationReport: {
        if(!isMaster) {
            errorCounter = data.value<Index>();
            emit sendData(PT02errorEstimationReport);
        } else {
        }
        error = (double)errorCounter/boolList.size();
        emit logMessage(QString("#01: Estimated error rate: %1 / %2 (%3%)").
                        arg(errorCounter).arg(boolList.size()).arg(error*100));
        boolList.clear();
        k1 = calculateInitialBlockSize(error);
        emit logMessage(QString("#02: resulting block size (k1): %1").arg(k1));
        emit logMessage(QString("Error Estimation finished"), Qt::green);

        emit logMessage("Initial Parity Check started", Qt::green);
        Index maximumBlockSize = k1*pow(2,runCount-1);
        Index removeBits = reorderedMeasurements.first().size() % maximumBlockSize;
        Measurements::iterator end = reorderedMeasurements.first().end();
        reorderedMeasurements.first().erase(end-removeBits,end);
        Q_ASSERT(reorderedMeasurements.first().size()/(SIndex)maximumBlockSize > 0);

        runIndex = 0;
        transferedBitsCounter = 0;
        this->incomingData(PT03prepareBlockParities, (uint)runIndex);

        return;
    }

    case PT03prepareBlockParities: {
        if(data.userType() == idIndexList) {
            reorderedMeasurements.append(reorderMeasurements(
                reorderedMeasurements.last(), data.value<IndexList>()));
            runIndex = reorderedMeasurements.size() - 1;

        } else if(data.type() == (QVariant::Type)QMetaType::UInt) {
            runIndex = data.toUInt();
        }
        Q_ASSERT(runIndex < reorderedMeasurements.size());

        blockSize = k1*pow(2,runIndex);
        binaryBlockSize = blockSize;
        blockCount = reorderedMeasurements.at(runIndex).size()/blockSize;
        emit logMessage(QString("in PT03prepareBlockParities, reorderedMeasurements.size = %1,"
                                "runIndex = %2, blockSize = %3").
                        arg(reorderedMeasurements.size()).arg(runIndex).arg(blockSize));
        parities.clear();
        Index lastIndex = reorderedMeasurements.at(runIndex).size() - blockSize;
        for(Index index = 0; index <= lastIndex; index += blockSize) {
            parities.append(calculateParity(reorderedMeasurements.at(runIndex),
                                            index, blockSize));
        }

        if(isMaster) {
            emit sendData(PT04reportBlockParities,
                        QVariant::fromValue<BoolList>(parities));
        }

        return;
```

```
366     }
367     case PT04reportBlockParities: {
368         if(!isMaster) {
369             qint64 size = parities.size();
370             Q_ASSERT(size > 0);
371             BoolList compareParities = data.value<BoolList>();
372             Q_ASSERT(compareParities.size() == size);
373             corruptBlocks.clear();
374             for(Index index = 0; index < size; index++) {
375                 if(compareParities.at(index) != parities.at(index))
376                     corruptBlocks.append(index*blockSize);
377             }
378             emit sendData(PT04reportBlockParities,
379                           QVariant::fromValue<IndexList>(corruptBlocks));
380         } else {
381             corruptBlocks = data.value<IndexList>();
382         }
383         transferedBitsCounter += corruptBlocks.size();
384         if(blockSize == binaryBlockSize) {
385             emit logMessage(QString("number of corrupt Blocks: %1/%2 (%3%)").
386                             arg(corruptBlocks.size()).
387                             arg(blockCount).
388                             arg((double)corruptBlocks.size()/blockCount*100));
389         }
390
391         if(isMaster) {
392             if(corruptBlocks.empty()) {
393                 runIndex++;
394                 // startBinary finished and there is another run
395                 // to return to (orders.size > 0)
396                 if((binaryBlockSize == 1) && (reorderedMeasurements.size() > 1) &&
397                    (runIndex == reorderedMeasurements.size())) {
398                     runIndex = 0;
399                 }
400                 if(runIndex < reorderedMeasurements.size()) {
401                     emit sendData(PT03prepareBlockParities, (uint)runIndex);
402                     this->incomingData(PT03prepareBlockParities);
403                 } else if(reorderedMeasurements.size() < runCount) {
404                     QVariant data = QVariant::fromValue<IndexList>(
405                         getRandomList(reorderedMeasurements.last().size()));
406                     emit sendData(PT03prepareBlockParities, data);
407                     this->incomingData(PT03prepareBlockParities, data);
408                 } else {
409                     emit sendData(PT06finished);
410                 }
411             } else { // start Binary
412                 this->incomingData(PT05startBinary);
413             }
414         }
415         return;
416     }
417
418     case PT05startBinary: {
419         binaryBlockSize = binaryBlockSize/2;
420         parities.clear();
421         if(isMaster || (!isMaster && binaryBlockSize > 1)) {
422             foreach(Index index, corruptBlocks) {
423                 parities.append(calculateParity(reorderedMeasurements.at(runIndex),
424                                                 index, binaryBlockSize));
425             }
426         }
427
428         if(isMaster) {
429             emit sendData(PT05startBinary,
430                           QVariant::fromValue<BoolList>(parities));
431         } else {
432             BoolList compareParities = data.value<BoolList>();
433             Index size = compareParities.size();
434             Q_ASSERT(corruptBlocks.size() == (SIndex)size);
435             if(binaryBlockSize > 1) {
436                 Q_ASSERT(parities.size() == (SIndex)size);
437                 for(Index i = 0; i < size; i++) {
438                     if(compareParities.at(i) == parities.at(i)) {
439                         // the 2nd half of the block must me corrupt
```

```
                    corruptBlocks.replace(i,corruptBlocks.at(i)+binaryBlockSize);
                }
            }
        } else { // binaryBlockSize == 1: just toggle wrong bits
            for(Index i = 0; i < size; i++) {
                // position in reorderedMeasurements: bit1 | bit2
                // compareParities contains bit1 from Alice (master)
                bool &bit1 = reorderedMeasurements.at(runIndex).
                        at(corruptBlocks.at(i))->bit;
                bool &bit2 = reorderedMeasurements.at(runIndex).
                        at(corruptBlocks.at(i)+1)->bit;
                if(compareParities.at(i) == bit1) {
                    bit2 = !bit2;
                    // corruptBlocks.replace(i, corruptBlocks.at(i)+1);
                }
                bit1 = compareParities.at(i);
            }
            corruptBlocks.clear();

        }
    }
    if(!isMaster) {
        emit sendData(PT04reportBlockParities,
                      QVariant::fromValue<IndexList>(corruptBlocks));
        transferedBitsCounter += corruptBlocks.size();
    }
    return;
}
case PT06finished: {
    if(!isMaster) {
        emit sendData(PT06finished);
    } else {
        this->sendData(PT07evaluation);
    }
    transferedBitsCounter += reorderedMeasurements.last().size()/
                k1*(2-pow(2,1-runCount));
    emit logMessage(QString("bitCounter: %1 (%2%)").arg(transferedBitsCounter).
                    arg(100.0*transferedBitsCounter/reorderedMeasurements.last().size()));

    // An increasing security parameter s lowers the upper bound of Eve's
    // information on the key I <= 2^(-s)/ln(2); 2^(-7)/ln(2) = 0.01 bits
    const quint8 securityParameter = 7;
    qreal removeRatio = error*2
                    + (qreal)(transferedBitsCounter+securityParameter)/
                                reorderedMeasurements.last().size();
    if(!(removeRatio < 1.0)) {
        emit logMessage(QString("too much information revealed for privacy-amplification: %1%").
                        arg(removeRatio*100), Qt::red);
        return;
    }

    QByteArray finalKey = privacyAmplification(reorderedMeasurements.last(), 1-removeRatio);
    emit logMessage(QString("finished! (%1 byte)").arg(finalKey.size()));
    {
        QFile file(QString("outfile_%1.dat").arg(isMaster ? "alice" : "bob"));
        file.open(QIODevice::WriteOnly);
        QDataStream out(&file);
        out << finalKey;
        file.close();
    }
    int imageSize = qFloor(qSqrt(finalKey.size()/3));
    QImage image(imageSize, imageSize, QImage::Format_RGB888);
    int pos = 0;
    for(int x = 0; x < imageSize; x++) {
        for(int y = 0; y < imageSize; y++) {
            image.setPixel(x,y, qRgb(finalKey.at(pos),
                                     finalKey.at(pos+1),
                                     finalKey.at(pos+2)));
            pos+=3;
        }
    }
    image.save(QString("outfile_%1.png").arg(isMaster ? "alice" : "bob"));
    emit imageGenerated(image);
    return;
```

```
514     }
515
516     case PT07evaluation: {
517         parities.clear();
518         Index size = reorderedMeasurements.last().size();
519         for(Index index = 0; index < size; index++)
520             parities.append(calculateParity(reorderedMeasurements.last(), index, 1));
521         if(!isMaster) {
522             emit sendData(PT07evaluation, QVariant::fromValue<BoolList>(parities));
523         } else {
524             BoolList compareParities = data.value<BoolList>();
525             Q_ASSERT(compareParities.size() ==(SIndex)size);
526             Index errors = 0;
527             for(Index index = 0; index < size; index++) {
528                 if(parities.at(index) != compareParities.at(index))
529                     errors++;
530             }
531             emit logMessage(QString("Ergebnis: %1 (%2%) Bit-Fehler").arg(errors).
532                             arg(100.0*errors/size));
533         }
534
535         emit finished();
536
537         return;
538     }
539
540     }
541 }
542
543 void QKDProcessor::start()
544 {
545     if(isMaster) { // this is Alice
546         // we kindly ask Bob for a list a received bits in first range from 0-quint32
547         emit sendData(PT01sendReceivedList);
548     } // Bob doesn't do anything here
549 }
```

Float Diagram for Error Correction

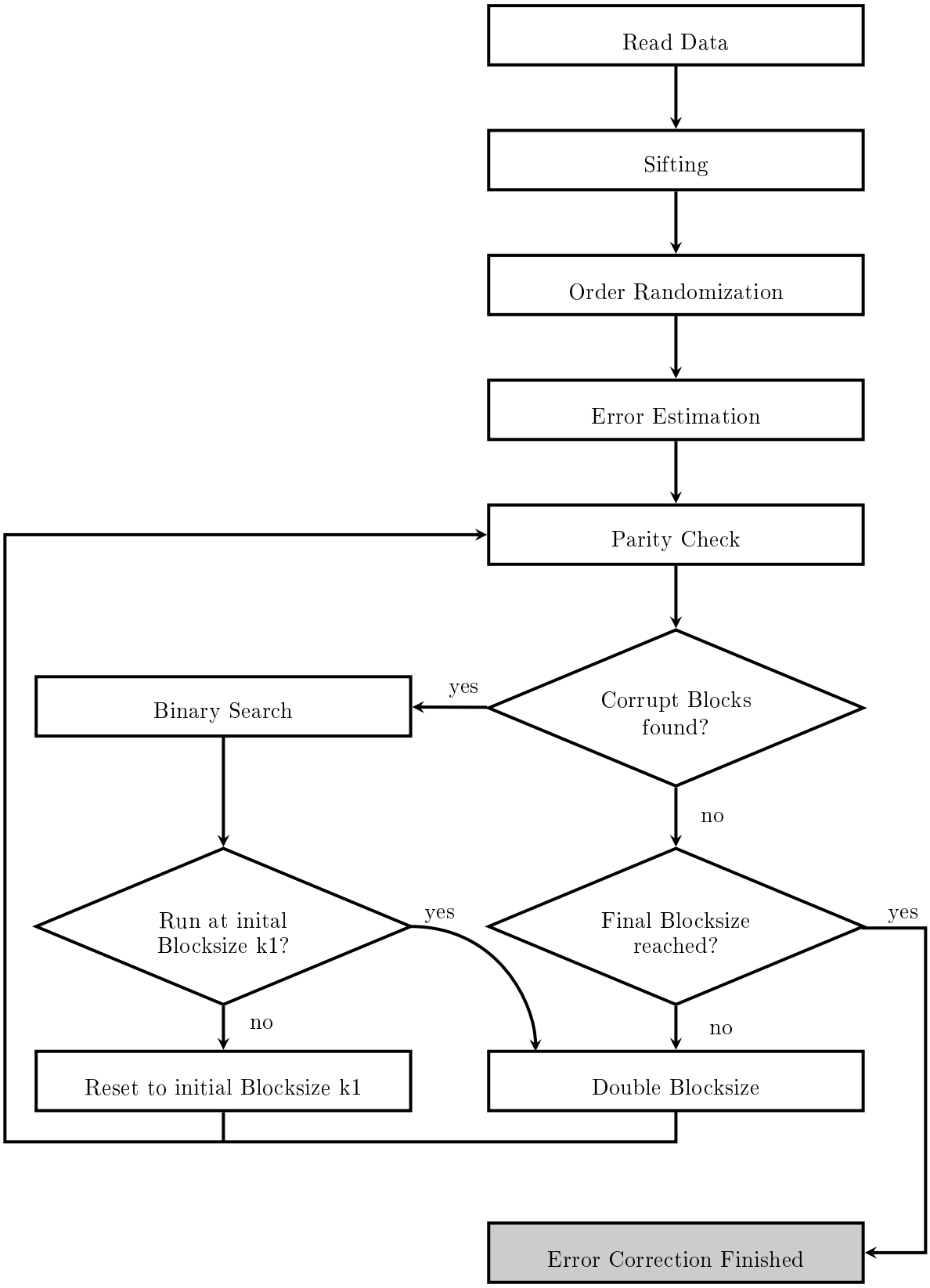

Figure 2: Float Diagram for Error Correction