A Quantum Network Manager That Supports A One-Time Pad Stream

Alan Mink, Lijun Ma, Tassos Nakassis, Hai Xu, Oliver Slattery, Barry Hershman and Xiao Tang
Information Technology Laboratory,
National Institute of Standards and Technology (NIST),
100 Bureau Dr., Gaithersburg, MD 20899
amink@nist.gov

Abstract

We have begun to expand the NIST quantum key distribution (QKD) system into a quantum network to support secure cryptography. We are starting with a simple three-node network, one Alice switched between Bob1 and Bob2. To support such a quantum network, we have implemented a quantum network manager that not only handles the switch and QKD protocol startup operations but also handles multiplexing and synchronization of secret key streams. We describe the function, structure and interfaces of this quantum network manager and report on initial switching overhead. We also discuss some steps we plan to take to optimize that overhead as well as hide its latency for certain applications.

1. Introduction

The NIST Quantum Information program has produced a number of record setting quantum key distribution (QKD) systems [1,10,12]. These point-to-point QKD systems do not themselves send secure messages. Rather, a QKD system develops secure keys between two end points over unsecured channels that can then be used to encrypt messages that are sent over open networks, such as the Internet. Similarly a quantum network is not a network to send secure messages, it is a network through which one can develop secure keys. Extending QKD systems into a network is the next step towards deployment feasibility.

A number of researchers discuss quantum networks. Jackson, et. al. [5] outline the need for a quantum network. They propose to develop quantum network architecture and associated protocols, suggesting the solution depends on the ability to delay arriving photons. Curcic, et. al. [2] also discuss the need for a quantum network but focus more on quantum repeaters. BBN [3, 4] has implemented a quantum network to support QKD based cryptography. Four of the BBN nodes, two Alice and Bob pairs, use different technologies and are not switchable. They use a hop-by-hop key transport to provide end-to-end keys. Two other Alice and Bob pairs use compatible technology and are networked via a Micro-Electro-Mechanical System (MEMS) switch. BBN uses VPN tunnels through IPsec but does not mention a quantum network manager.

Even if IPsec is the only utility using the QKD network and all applications do their encryption/decryption through IPsec, a quantum network manager is necessary to manage the QKD switching fabric and any associated Alice/Bob initialization. It can also provide multiplexing of independent key streams while keeping them synchronized. This is a necessary function that would otherwise have to be handled by IPsec, especially if one-time pad encryption is offered. If these key streams lose synchronization, then we no longer have a shared secret that can be used to successfully decrypt messages. We present an outline of our initial implementation of a quantum network manager that can provide this support. We start off with a brief overview of our QKD network configuration.

2. QKD configuration and startup

Our point-to-point QKD configuration, as shown in Fig 1, is discussed in [1,7,10,12]. Although we have successfully used a single fiber to wavelength division multiplex (WDM) the quantum channel and the classical channel, we have found there is some WDM mode mixing that adds noise to both channels resulting in a higher quantum error rate. As a result we have been using separate fibers for the two channels to minimize the error rate and thus maximize our overall secret key generation rate. The associated QKD protocol stack is shown in Fig 2. The raw key stream generation and management along with the sifting algorithm is implemented in hardware, a custom printed circuit board (PCB). This hardware allows us to send photons at GHz to retrieve Mb/s of sifted key, which is a more manageable rate for the remainder of our protocol stack implemented in software. Our enhanced software versions of the Cascade reconciliation algorithm and the privacy amplification algorithm have the capacity to yield a few Mb/s of secret key. Planned enhancements to our custom PCB [8] will allow us to migrate these software algorithms to hardware with a capacity of between 10 and 20 Mb/s of secret key. The output from our QKD protocol stack is an ordered set of identical bits (secret keys) at both Bob and Alice.

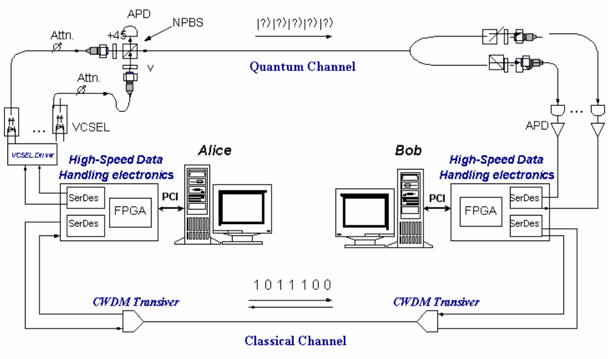


Fig. 1. NIST QKD system configuration.

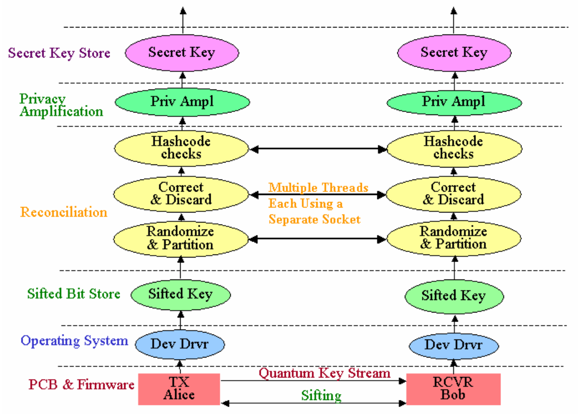


Fig. 2. NIST QKD infrastructure protocol stack.

A quantum network manger needs to handle startup procedures. Our QKD system requires polarization recovery and compensation, and quantum-to-classical channel timing alignment. Polarization recovery and compensation [6] is necessary because polarization changes occur as a photon transits through a fiber optic cable. This procedure recovers the current polarization drift for each independent polarization base (both BB84 and B92 have two independent bases) and adjusts piezo-driving polarization controllers to compensate for that drift. Once this procedure is completed the optics are operationally ready. Different photon encoding schemes may require different start-up procedures that may vary in complexity and time.

Quantum-to-classical channel timing alignment is necessary because we use the classical channel to establish our timing reference points and to recover the transmit clock frequency. Even when the classical and quantum channels share a common fiber, their data paths are different and we need to compensate for those differences. On the PCB we have independent sub-bit timing alignment to compensate for any phase difference between each quantum channel and the classical channel. Within the programmable chip (an FPGA – Field Programmable Gate Array) on the PCB, we have independent multiple bit timing alignment to compensate for any delays each quantum bit stream may encounter.

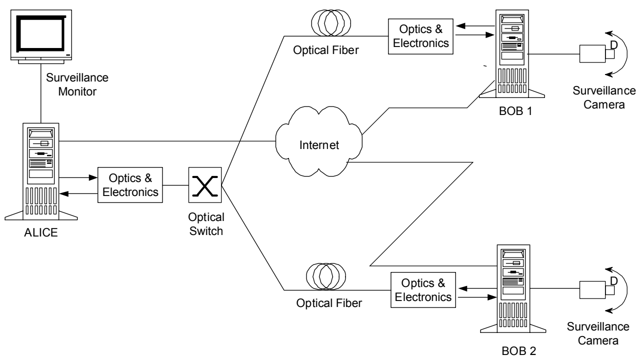


Fig. 3. NIST QKD network configuration.

As a first extension towards a quantum network we've added an additional Bob and a pair of MEMS optical switches, one for the classical channel and the other for the quantum channel. These switches are configured to operate together simultaneously. When configured to share a single fiber for both channels, only a single switch is needed. Fig 3 shows both the switch on the quantum network that develops the secret keys and the Internet over which the encrypted messages travel.

3. Network manager functionality

A quantum network manager needs to set the switch as directed, complete any QKD start-up procedures, start the QKD protocols to enable the secret key flow and then stop the protocols to disable the flow when switching is requested. Providing the synchronization and multiplexing of independent secret key streams between both source and destination is also beneficial.

To support synchronized multiplexing we use the concept of a FIFO. Each user can operate one or more FIFOs, where a FIFO contains an independent stream of secure keys. A duplicate FIFO exists at both the source and destination. The FIFO pairs are filled with the same amount and the same sequence of secure keys. For example, an application on Alice and Bob1 can each open two FIFO pairs, one pair for encrypting data to be sent to Bob1 and the other pair for decrypting data received from Bob1. If the application also existed on Bob2, then another two FIFO pairs could be used to encrypt/decrypt messages between Alice and Bob2. This way each message stream has its own FIFO pair to use for secret keys and incurs no additional overhead in trying to synchronize the keys among the messages. Otherwise both source and destination applications would have to coordinate and reserve the same key material for each message.

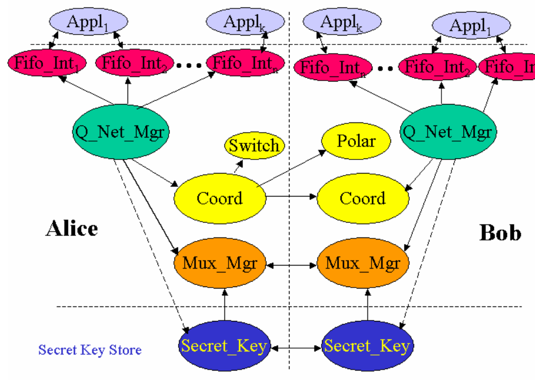


Fig. 4. Quantum Network Manager structure.

Our quantum network manager, depicted in Fig 4, consists of a main loop that spawns a number of threads [11] as well as our QKD protocol stack as a separate process The coordination manager thread (*COORD_Mgr*) controls various equipment such as the switch, the polarization compensation module and the manager at the other end of quantum network link. The FIFO multiplex manager (*MUX_Mgr*) synchronizes the key flow from the secret key store of our QKD protocol stack into the various FIFOs, as illustrated in Fig 5. It uses a separate socket to exchange information with its cooperating counterpart at the other end of the quantum network link about which keys are to be added to each FIFO. For each FIFO, the manager spawns a FIFO interface thread (*Fifo_Int*) which provide the user interface between the application and the FIFO.

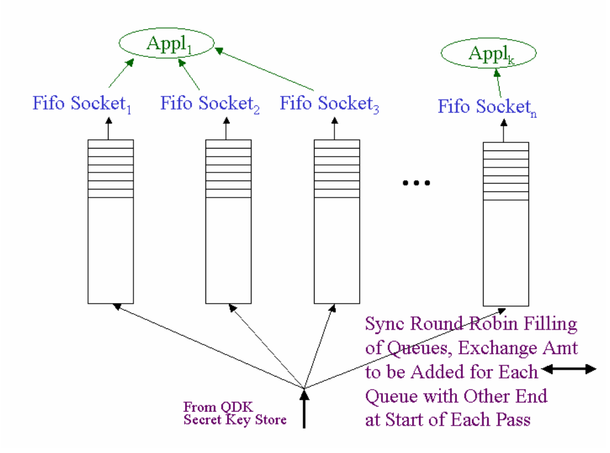

Appl1
Applk
Fifo Socket1
Fifo Socket2
Fifo Socket3
Fifo Socketn
Sync Round Robin Filling of Queues, Exchange Amt to be Added for Each Queue with Other End at Start of Each Pass
From QDK Secret Key Store

Fig 5. The MUX_Mgr synchronizes keys entering all FIFOs.

The code skeleton of the quantum network manager is shown in Fig 6. The main loop functions as the Open operation for the quantum network and accomplishes two tasks. It reserves a FIFO (*Fifo_ID*) and establishes a socket interface (*sock_fd*) between the application and that FIFO. An application can open a number of instances of the quantum network. Each instance is associated with a different FIFO.

```
Spawn COORD_Mgr Thread

Spawn MUX_Mgr Thread

While (FOREVER) {
   sock_fd = Wait_for_Appl2Open_Q_Net
   if (Fifo_ID <  0)
          Fifo_ID = assign_new_Fifo_ID
   else
          Process_existing_Fifo_ID
   if (Fifo_ID == VALID)
          Spawn FIFO_Interface Thread
   else
          close (sock_fd)
}
```

Fig 6. Skeleton of quantum network manager.

We use sockets as our communications interface, since the quantum network manager is a distributed application and doesn't need to exist on the same machine as the applications that uses it or the utilities that it invokes. This implies that the manager and its calling applications must exist within a secure enclave. All applications and the manager related to Alice are in one secure enclave while those associated with Bob1 are within a separate secure enclave.

As an example, one side of an application opens the quantum network asking for a new *Fifo_ID*. It then sends a message to the other sides of the application to use this newly reserved *Fifo_ID*. Then the other sides of the application opens the quantum network with that *Fifo_ID* value, allowing that application interface to be associated with the same FIFO as its cooperating application. This is similar to opening a socket with one side "listening" while the other side "connects". The *Fifo_ID* functions similarly to the socket port number to provide a common link between the cooperating applications, except here it is initially assigned by the quantum network rather than by the application.

4. Key synchronization loss and recovery

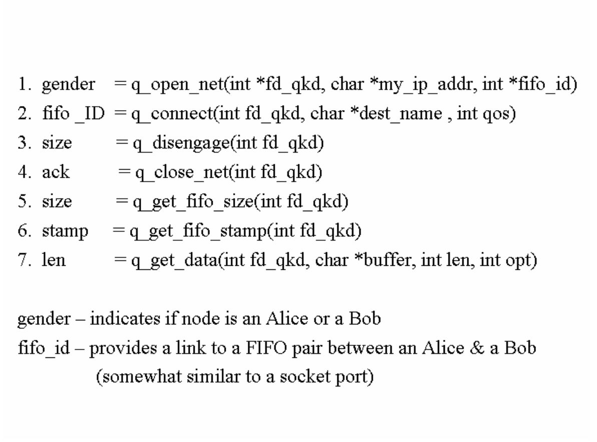
Since duplicate secret keys are kept at the respective ends of the channel, there is always a concern the some bits may be dropped or corrupted. A few corrupted bits will ruin a key, but future keys will not be affected. A few dropped bits (loss of key synchronization) however, can be disastrous. Since the keys exist as an ordered set of random bits duplicated at each end, if any are dropped then all future key streams will differ at their respective ends and all future encrypted messages will be undecodeable.

Recovery from a few corrupted bits can be accomplished by discarding that key and obtaining a new key. Preventing such an occurrence can be accomplished by exchanging an error detection (hash or CRC) code for each group of keys transferred at each layer. This incurs some additional overhead and also requires that some bits of the key be discarded to eliminate information potentially revealed by the hash code. For example, if 1 Mbit of key is transferred from the privacy amplification layer to this network manager and a 64-bit error detection code is exchanged to verify their equivalence, then 64-bits of key will need to be discarded to compensate for any revealed information about the secret keys. If the codes don't match, then that entire group of keys will be discarded.

At each layer, steps are taken to prevent loss of key synchronization. At the reconciliation layer, if the duplicate sets of bits don't match, they are discarded. If high error rates are detected, then our custom PCB is reset and restarted. This detects and corrects both corrupted and dropped bits. In the privacy amplification layer each byte of the secret key is assigned a sequential value. When a group of bits is transferred out of the privacy amplification layer, the sequential value associated with the first byte is also sent, called an ID stamp. The receiving process, in this case the *MUX_Mgr* of the quantum network manager, knows the ID stamp and the amount of the previous group of keys it received, so it can compute the expected ID stamp of the next group of keys. The *MUX_Mgrs* also exchange their respective ID stamps to verify synchronization. If they don't match, the keys will be discarded and the entire protocol stack will be reset and restarted in coordination with the cooperating *MUX_Mgr*. An alternative to this draconian reset and restart strategy is to assume that key was lost in the transfer of data but that the ID stamps are correct. In that case, the *MUX_Mgrs* can coordinate a strategy that only requires the two sides to discard the last group of keys and use the ID stamp to find a new synchronization point in the key stream. The concept of an ID stamp associated with the secret key is continued at the quantum network manager layer, which provides a separate, independent ID stamp for the keys in each FIFO for all of its user processes.

5. Quantum Network APIs

The application program interface (API) functions for the quantum network manager are listed in Fig 7. The *q_open_net* function reserves a FIFO and establishes a socket interface between the application and that FIFO. The FIFO is not allocated until the initial connection is established. This is accomplished through the *q_connect* function. Through the "qos" parameter, an application can specify the class of keys desired (e.g., a few kbits for a standard AES encryption or a large stream of bits for a one-time pad). In addition to allocating the FIFO, *q_connect* sets the switch, runs the polarization compensation setup and initiates the QDK protocol stack, if not already active. If the network is already connected, we only verify that the request is for this current connection, otherwise the request fails. We have no restrictions as to who may set the switch. The only requirement is that no one is currently connected when a request is made. The *q_disengage* function allows an application to signal that it is temporarily finished with the current connection for a specific FIFO and that no new secret keys should be added to that FIFO. Although the *Mux_Mgr* handles the synchronization and assures that the same keys are put into a FIFO, the application is responsible for keeping track of the keys it takes out of a FIFO. The *q_disengage* function is a prerequisite for switching the network or permanently closing a FIFO. Only when all FIFOs are disengaged do we allow switching to occur. The application can still get data and information on a FIFO when it has been disengaged. Only the operations associated with filling a FIFO stop once its disengaged. The *q_close_net* function permanently closes and deallocates a FIFO, but first it must have been disengaged. Any secret keys remaining in that FIFO are discarded. The FIFO is now eligible for reassignment to another user.



```
1. gender   = q_open_net(int *fd_qkd, char *my_ip_addr, int *fifo_id)
2. fifo _ID = q_connect(int fd_qkd, char *dest_name , int qos)
3. size     = q_disengage(int fd_qkd)
4. ack      = q_close_net(int fd_qkd)
5. size     = q_get_fifo_size(int fd_qkd)
6. stamp    = q_get_fifo_stamp(int fd_qkd)
7. len      = q_get_data(int fd_qkd, char *buffer, int len, int opt)

gender – indicates if node is an Alice or a Bob
fifo_id – provides a link to a FIFO pair between an Alice & a Bob
           (somewhat similar to a socket port)
```

Fig 7. Quantum Network Manager APIs.

Three API functions provide data and information to the application. The *q_get_fifo_size* function is a non-blocking function and returns the number of bytes of secret keys available in the FIFO. The *q_get_fifo_stamp* function is a non-blocking function and returns the number of bytes read out of the FIFO and is meant to be used by the application to identify specific keys. If an application uses keys in other than sequential order or if it randomly decides to change keys, it can send the current FIFO stamp to its peer to establish the keys it is using. The *q_get_data* function places secret keys from the FIFO in the buffer and returns the number of bytes it has placed in the buffer. This is normally a blocking function but can be optionally invoked as a non-blocking function.

6. Test results

We have implemented two test applications for this quantum network manager, a simple test application and a video application. Our simple test application establishes two FIFOs at each node, one for a transmit stream and the other for a receive stream. Initially Alice connects to Bob1. They both send 1 Kbyte of keys as a message to each other. Alice sends from FIFO "0" and Bob1 sends from FIFO "1". Upon receipt of the message they read the keys from the appropriate FIFO and compare them against what was received and then print out an accumulated error count. This count should always be zero. They then check how full the FIFO is, get the "stamp" value, and then disengage to prepare for switching. After disengaging, they partially empty their FIFOs and Alice switches to Bob2 and repeats this same sequence of operations. Alice continues to switch between Bob1 and Bob2 executing these operations until it is stopped.

Table 1. Initial Switching Time Distribution

Operation	Time (sec)
Switch	.001
Polarization Compensation	40
Board Alignments	5
Initial Key Acquisition Delay	50
Total	95

For this test application we have measured the circuit switching time and how that time is distributed. Our results are shown in Table 1. The switching time between Bob1 and Bob2 is less than 1 ms. Our polarization compensation procedure takes about 40 sec. Alignment for our custom printed circuit board takes about 5 sec but this only occurs once, the first time we connect to a node. Lastly there is the delay as we acquire raw keys and execute the QKD protocol. This currently takes about 50 sec, because we operate on multi-Mbit chunks of raw keys, yielding about 16 Mbits of secret keys before our initial deposit to the secret key store. After this initial delay, our deposits to the key store average between 0.25 and 1.1 Mb/s depending on our operating conditions.

Our simple test application is used for functional testing and no effort has been made to optimize or hide the latency of these switching operations. We have developed a video application that encrypts webcam video in real time using a one-time pad [9]. We have employed strategies to optimize and hide the switching overhead by continuing to send encrypted video from Bob1, while the switching begins. Since the FIFO is disengaged, we rely on its accumulated key to prevent stall of video. We will modify our QKD protocols to operate on smaller amounts of raw key, resulting in faster deposits of secret key in smaller chunks. During the initial key acquisition we change the video flow from Bob1 to Bob2, and depending on the remaining key left in Bob2's FIFO, we can start sending encrypted video before we start getting new secret key. Although this does not completely eliminate the switching delay, it reduces it and hides a large portion of it.

7. Summary

We have begun to expand our QKD system into a quantum network to support secure cryptography. We have started with a simple three-node network, one

Alice switched between Bob1 and Bob2. To support that quantum network, we have implemented a quantum network manager that not only handles the switch and QKD protocol startup operations but also handles multiplexing and synchronization of secret key streams. We have described the function, structure and interfaces of this quantum network manager and have reported on its initial switching overhead. We also discussed some steps we plan to take to optimize that overhead as well as hide its latency for certain applications.

An additional benefit of a quantum network manager that provides independent synchronized key streams is that it can support multiple users/applications. As an example, it could simultaneously support IPsec, Transport Layer Security and a number of other independent applications. Although we present a simple FIFO interface to users, something more complex, like a file interface that supports random access, may be more appropriate if the application needs to handle packet assembly and out-of-order packets.

Acknowledgement

This work was partially supported by the Defense Advanced Research Projects Agency under the DARPA QuIST program and by NIST under the Quantum Information Initiative program.

References

[1] J. C. Bienfang, A.J. Gross, A. Mink, B.J. Hershman, A. Nakassis, X. Tang, R. Lu, D.H. Su, C.W. Clark, C.J. Williams, E.W. Hagley, and J. Wen, "Quantum key distribution with 1.25 Gbps clock synchronization," Optics Express, Vol. 12 (9), 2011 (2004). <http://www.opticsinfobase.org/abstract.cfm?URI=oe-12-9-2011>.

[2] T. Curcic, E. Flilipkowski, A. Chtchelkanova, P. D'Ambrosio, S. Wolf, M. Foster and D. Cochran, "Quantum Networks: From Quantum Cryptography to Quantum Architecture", ACM SIGCOMM Computer Communications Review, Vol. 34, No 5, Oct 2004. <http://portal.acm.org/citation.cfm?doid=1039117>.

[3] C. Elliott, "Building the quantum network", New J. of Phys, 4 (2002), 46. <http://www.iop.org/EJ/abstract/1367-2630/4/1/346>.

[4] C. Elliott, A. Colvin, D. Pearson, O. Pikalo, J. Schlafer and H. Yeh, "Current Status of the DARPA Quantum Network", Mar. 2005. <http://arxiv.org/ftp/quant-ph/papers/0503/0503058.pdf>.

[5] D. Jackson, D. Giliam, J.Dowling, "Quantum network protocols", Presented at the National Society of Black Physicists Annual Day of Lectures Held Jointly with National Conference of Black Physics Students, Palo Alto, CA, Mar 2001 <http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/12536/1/01-0726.pdf>, <http://hdl.handle.net/2014/12536>.

[6] L. Ma, H. Xu, and X. Tang, "Polarization recovery and auto-compensation in Quantum Key Distribution network," Proc. of SPIE, Aug. 2006, Vol. 6305, pp. 630513. <http://w3.antd.nist.gov/pubs/2007/Polarization recovery and auto-compensation.pdf>.

[7] A. Mink, X. Tang, L. Ma, T. Nakassis, B. Hershman, J. Bienfang, D. Su, R. Boisvert, C. Clark, and C. Williams, "High Speed Quantum Key Distribution System Supports One-Time Pad Encryption of Real-Time Video," Proc. of SPIE Defense & Security Symposium, Orlando, FL, 17-21 April 2006, Vol. 6244, pp. 62440M 1-7. <http://w3.antd.nist.gov/pubs/Mink-SPIE-One-Time-Pad-6244_22.pdf>.

[8] A. Mink, ""Custom Hardware to Eliminate Bottlenecks in QKD Throughput Performance", Proc. of SPIE Optic East: Next-generation Photonic Sensor Technologies, Vol. .6780, Boston, MA, 8-12 Sept 2007, pp. 678014-1.

[9] One-time pad, <http://en.wikipedia.org/wiki/One-time_pad>, accessed Aug. 2007.

[10] X. Tang, L. Ma, A. Mink, T. Nakassis, H. Xu, B. Hershman, J. Bienfang, D. Su, R. Boisvert, C. Clark, and C. Williams, "Quantum Key Distibution System Operating at Sifted-key Rate Over 4 Mbits/s," Proc. of SPIE Defense & Security Symposium, Orlando, Vol. 6244-25, FL, 17-21 April 2006, pp. 62440P-1-7. <http://w3.antd.nist.gov/pubs/Xiao-SPIE-QKD-4mMbps-6244_25.pdf>.

[11] Threads, <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>, accessed Aug 2007.

[12] H. Xu, L. Ma, A. Mink, B. Hershman, and X. Tang "1310-nm quantum key distribution system with up-conversion pump wavelength at 1550 nm", Optics Express, June 11, 2007, Vol. 15, No. 12, pp 7247. <http://www.opticsexpress.org/abstract.cfm?id=138139>.