A Protocol for Synchronizing Quantum-Derived Keys in IPsec and its Implementation

Stefan Marksteiner

JOANNEUM RESEARCH GmbH
DIGITAL - Institute for Information
and Communication Technologies
Graz, Austria
Email: stefan.marksteiner@joanneum.at

Oliver Maurhart

AIT Austrian Institute of Technology GmbH
Digtal Safety & Security Department
Klagenfurt, Austria
Email: oliver.maurhart@ait.ac.at

***Abstract*—This paper presents a practical solution to the problem of limited bandwidth in Quantum Key Distribution (QKD)-secured communication through using rapidly rekeyed Internet Protocol security (IPsec) links. QKD is a cutting-edge security technology that provides mathematically proven security by using quantum physical effects and information theoretical axioms to generate a guaranteed non-disclosed stream of encryption keys. Although it has been a field of theoretical research for some time, it has only been producing market-ready solutions for a short period of time. The downside of this technology is that its key generation rate is only around 12,500 key bits per second. As this rate limits the data throughput to the same rate, it is substandard for normal modern communications, especially for securely interconnecting networks. IPsec, on the other hand, is a well-known security protocol that uses classical encryption and is capable of exactly creating site-to-site virtual private networks. This paper presents a solution which combines the performance advantages of IPsec with QKD. The combination sacrifices only a small portion of QKD security by using the generated keys a limited number of times instead of just once. As a part of this, the solution answers the question of how many data bits per key bit make sensible upper and lower boundaries to yield high performance while maintaining high security. While previous approaches complement the Internet Key Exchange protocol (IKE), this approach simplifies the implementation with a new key synchronization concept. Furthermore, it provides a Linux-based module for the AIT QKD software using the Netlink XFRM Application Programmers Interface to feed the quantum key to the IPsec cipher. This enables wire-speed, QKD-secured communication links for business applications.**

Keywords–Quantum Key Distribution; QKD; IPsec; Cryptography; Security; Networks.

I. Introduction and Motivation

Quantum cryptography, in this particular case quantum key distribution (QKD), has the purpose to ensure the confidentiality of a communication channel between two parties. The main difference to classical cryptography is that it does not rely on assumptions about the security of the mathematical problem it is based on, nor the computing power of a hypothetical attacker. Instead, QKD presents a secure method of exchanging keys by connecting the two communicating parties with a quantum channel and thereby supplying them with guaranteed secret and true random key material [1, p.743]. When the key is applied with a Vernam cipher (also called one time pad - OTP) on a data channel on any public network, this method provides the channel with information-theoretically (in other words mathematically proven) security [2, p.583]. The downside of this combination is the limitation to approximately twelve kilobits, shown in a practical QKD setup, due to physical and technical factors, since with OTP one key bit is consumed by one data bit [3, S.9]. This data rate does not meet the requirements of modern communications. Another practical approach came to the same conclusion and therefore uses the Advanced Encryption Standard (AES) instead of OTP [4, p.6]. To address this problem, this paper presents an approach to combine QKD with IPsec, a widespread security protocol suite that provides integrity, authenticity and confidentiality for data connections [5, p.4], by using QKD to provide IPsec with the cryptographic keys necessary for its operation. To save valuable key material, this solution uses it for more than one data packet in IPsec, thus increasing the effective data rate, which is thereby not limited to the key rate anymore. Furthermore, using this approach, the presented solution benefits from the flexibility of IPsec in terms of cryptographic algorithms and cipher modes. In contrast to most of the previous approaches, that supplemented the Internet Key Exchange (IKE) protocol or combine in some way quantum-derived and classical keys, this paper refrains from using IKE (for a key exchange is rather the objective of QKD, as described later) in favor of a specialized, lightweight key synchronization protocol, working with a master/slave architecture. The goal of this protocol is to achieve very high changing rates of purely quantum-derived keys on the communicating peers while maintaining the keys synchronous in a very resilient manner, which means to deal with suboptimal networking conditions including packet losses and late or supplicate packets. In order to fulfill this objective, the following questions need to be clarified:

- What is the minimum acceptable frequency of changing the IPsec key that will ensure sufficient security?
- What is the maximum acceptable frequency of changing the IPsec key to save QKD key material?
- Is the native Linux kernel implementation suitable for this task?
- How can key synchronicity between the communication peers be assured at key periods of 50 milliseconds and less?

As a proof of concept, this paper further presents a software solution, called QKDIPsec, implementing this approach in

 ISBN: 978-1-61208-431-2

C++. This software is intended to be used as an IPsec module for the multi platform hardware-independent AIT QKD software, which provides already a market-ready solution for OTP-based QKD. The module achieves over forty key changes per second for the IPsec subsystem within the Linux kernel. At present time, the software uses a static key ring buffer for testing purposes instead of actual QKD keys, for the integration of QKDIPsec into the AIT QKD software is yet to be implemented (although most of the necessary interfaces are already present). The ultimate goal is to deliver a fully operational IPsec module for the AIT QKD software.

Section II of this paper describes considerations regarding necessary and sensible key change rates, while Section III contains the architecture of the presented solution and the subsequent Section IV its implementation. Section V, eventually, contains testing results and the conclusions drawn.

II. KEY CHANGE RATE CONSIDERATIONS

The strength of every cryptographic system relies on the key length, the secrecy of the key and the effectiveness of the used algorithms [6, p.5]. As this solution relies on QKD, which generates a secret and true random key [7], this means that more effective algorithms and more key material are able to provide more cryptographic security. In this particular case, the used algorithms are already prescribed by the IPsec standard [8], therefore the security is mainly determined by the used key lengths, more precisely by the relation between the amount of key material and the amount of data, which should be as much in favor of the key material as possible - given the low key rate compared to the data rate, naturally the opposite is the case in practice. This Section aims on giving feasible upper and lower boundaries of key change rates (or key periods P_k, respectively) and, thus, how much QKD key material should be used in order to save precious quantum key material while maintaining a very high level of security. The two main factors determining the key period in practice are the used algorithms (via their respective key lengths - the longer the key, the more key bits are used in one key period) and the capabilities of QKD in generating keys. Current working QKD solutions (such as the one used by the AIT) provide a quantum key rate Q of up to 12,500 key bits per second at close distances, 3,300 key bits at around 25 kilometers and 550 key bits at around 50 kilometers distance [3, p.9].

In order to fully utilize the possibly QKD key rate and given the currently shortest recommended key length, which is 128 bits, a IPsec solution using quantum-derived keys should thus be able to perform around 100 key changes per second ($\frac{12{,}500}{128} \approx 97,65$), 50 for every communication direction (for IPsec connection channels are in principle unidirectional and therefore independent from each other even if they belong to the same bidirectional conversation). This corresponds to a key period P_k of around 20 ms, as it is a function of the Quantum key rate Q and the algorithm's key length k. The period for a bidirectional IPsec link is $P_K = (\frac{Q}{2k})^{-1}$. At longer key lengths, this period becomes longer, for a single change cycle uses more key material and, thus, less key changes are necessary to utilize the full incoming key stream, therefore this period $P_{k_{min}} = 20ms$ presents a feasible lower boundary for the key period. As stated above, the security of this system depends also on the data rate. Given a widespread data rate of 100 megabits per second, a key period of 20 ms and 128 key bits means a ratio of 8000 data bits per key bit (or short dpk, for the reader's convenience).

A landmark in this *security ratio* is 1 dpk. This rate would provide unconditional security when applied with OTP. For the cipher and hash suites included in the IPsec protocol stack, there is no security proof and therefore they are not unconditionally secure. However, applying an IPsec cipher (for instance AES) with an appropriately fast key change and restricted data rate to achieve 1 dpk is the closest match inside standard IPsec, especially when the block size equals the key size.

To define an upper boundary (and therefore a minimum standard for the high security application of the presented solution), a very unfavorable relation between data and key bits through a high-speed connection of 10 gigabits of data is assumed. A recent attack on AES-192/256 uses $2^{69.2}$ computations with 2^{32} *chosen plaintext* [9, p.1]. Because of the AES block size of 128 bits, this corresponds to $2^{32} * 2^7 = 2^{39}$ data bits. Although this attack is currently not feasible in practice, as it works only for seven out of 12/14 rounds and also has unfeasible requirements to data storage on processing power for a cryptanalytic machine, it serves as a theoretical fundament for this upper boundary. A bandwidth of 10 gigabits per second equals approximately 9.3 gibitbs per second. This is by the factor of 64 (2^6) smaller than the amount of data for the attack mentioned above, which means that it requires 64 seconds to gather the necessary amount of data to (though only theoretically) conduct the attack. In conclusion (with AES-192/256), the key should be changed at least every minute ($P_{k_{max}} = 60s$), while the maximum allowed key period according to the IPsec standard lies at eight hours or 28,800 seconds [10].

For cryptographic algorithms operating with lower cipher block sizes (ω), the *birthday bound* ($2^{\frac{\omega}{2}}$) is relevant. The birthday bound describes the number of brute force attempts to enforce a collision with a probability of 50 percent, such that different clear text messages render to the same cypher text. With a block size of 64 (*birthday bound* = 2^{32}), the example speed of 10 gigabit per second above would lower the secure key period to under half a second. Because of this factor, using 64-bit ciphers is generally discouraged for the use with modern data rates[11, pp.1-3] (although the present rapid rekeying approach is able to cope with this problem). Regarding key lengths, 128 bits are recommended beyond 2031 [6, p.67] while key sizes of 256 bits provide *good protection* even against the use of Grover's algorithm in hypothetical quantum computers for this period [12, p.32].

III. RAPID REKEYING PROTOCOL

This Section describes the *rapid rekeying protocol*, the purpose of which is to provide to IPsec peers with QKD-derived key material and keep these keys synchronous under the low-key-period conditions (down to $P_{k_{min}} = 20ms$) stated in Section II.

This protocol pursues the approach that with QKD, there is no need for a classical key exchange (for instance with IKE). Relevant connection parameters (like peer addresses) are available a priori (before the establishment of the connection) in point-to-point connections, whereas keying material is provided by QKD, mostly obsoleting IKE. Furthermore, IPsec only dictates an automatic key exchange, not specifically

 ISBN: 978-1-61208-431-2

IKE [5, p.48] and a protocol that only synchronizes QKD-derived keys (instead of exchanging keys) is therefore deemed sufficient, yet compliant to the IPsec standard. Consequently, it is an outspoken objective to create a slender and simple key synchronization protocol to increase performance and reduce possible sources of error. Another objective for key synchronization is robustness in terms of resilience against suboptimal network environment conditions. The protocol described in this paper uses two channels for encrypted communication: an Authenticaton Header (AH)-authenticated control channel (amongst other tasks, signaling for key changes) and an Encapsulating Security Payload (ESP)-encrypted data channel to transmit the protected data (see Figure 1). The reason for the use of AH on the control channel is that it only contains non-secret information, while its authenticity is crucial for the security and stability of the protocol. The necessary *security policies (SPs)* for the IPsec channels remain constant during the connection. There are four necessary SPs, one data and one control SP for each direction. The complete software solution will, delivered by the AIT QKD software, contain additionally the quantum channel for key exchange and a *Q3P* channel, whereby the latter is another protocol that provides OTP-encrypted QKD point-to-point links. These two additional channels are outside of this paper's scope.

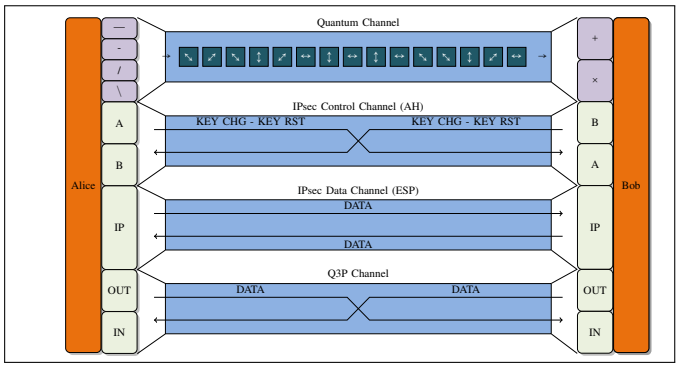

Figure 1. Rapid Rekeying Channel Architecture

The protocol itself follows, taking account of the unidirectional architecture of IPsec, a *master/slave* paradigm. Every peer assumes the master role for the connection in which the peer represents the sending part. When a key change is due (for instance because of the expiration of the key period), the master sends an according message (key change request) to the slave and the latter changes the key (as does the master). To compensate lost key change signals, every key change message contains the *security parameters index (SPI)* for the next-to-use key. The SPI is simply calculable for the peers through a salted hash whereby the salt and a initial seed value are QKD-derived and each SPI is a hash of its predecessor plus salt, which makes it non-obvious to third parties. This level of security is sufficient, for the SPI is a public value, included non-encrypted in every corresponding IPsec packet, making it a subject rather to non-predictability than to secrecy. Also, using only a seed and salt from QKD, the hashing method safes quantum keying material. As all necessary IPsec parameters are available beforehand, as well as the keys (through QKD), IPsec *security associations (SAs)* may be pre-calculated and established in advance (which are identified by unique SPIs). Permanently changing attributes during a conversation are only the SPI and the key, while all other parameters of an SA (for instance peer addresses, services, protocols) remain constant. The master calculates these two in advance and queues them for future use. Only one SA is actually installed (aplied to the kernel IPsec subsystem), for only one (per default, at least in Linux, the most recent) may be used to encrypt data. The slave, on the other hand, operates differently. For it identifies the right key to use based on the SPI, it may very well have multiple matching SAs installed. This makes key queuing expendable on the receiver side, while the SPI queuing is used as an indexer for lost key change message detection. For reasons of data packets arriving out of synchronization, SAs are not only installed beforehand, but also left in the system for some time on the receiver side, allowing it to process packets encrypted with both an older or newer key than the current one.

On every key change event, the master applies a new SA to the system (using the next following SPI/key from the queues), prepares a new SPI/key pair (SPI generation as mentioned above and acquirement of a new key from the QKD system) and deletes the deprecated data from both its queues and the IPsec subsystem. The slave also acquires a new SPI/key pair (the same the sender acquires) but installs it directly as an SA and only stores the SPI for indexing. It subsequently deletes the oldest SA from the system and SPI from the queue if the number of installed SAs exceeds a configured limit. To sum it up, on every key change event, the two peers conduct the following steps:

- the master acquires a new key and SPI and ads it to its queues
- it sends a key change request to the slave
- it fetches the oldest pair from the queue an installs it as a new SA, *replacing* the current one
- it deletes the deprecated pair from its queue
- the slave receives the key change request and also acquires a new SPI/key pair (the same as the master)
- it installs the pair as a new SA and the SPI into the indexing queue
- it deletes the oldest SA from the system and oldest SPI from the queue
- it sends a key change acknowledgement

This procedure keeps both of the installed SA types up to date. For instance, 50 installed SAs for the slave resulting in 25 queued SPI/key pairs on the master, for the latter does not need to store backward SAs. At the beginning, on every key change, SPI/key pair is acquired, while the already applied remain. When the (configurable) working threshold is met, additionally the oldest SA or SPI/key pair is deleted, keeping the queue sizes and number of installed SAs constant.

Figure 2 illustrates this process for a sender (*Alice*) and a receiver (*Bob*), where the arrows show the changes in case of an induced key change. Naturally, as with SPs, there are four SA types on a peer: one for data and control channels, each for sending (master) and receiving (slave). Each SA corresponds to an SPI and key queue on the master's side and one SPI queue on the slave's side, respectively.

As the data stream is independent from control signaling, this calculation in advance prevents the destabilization of the

 ISBN: 978-1-61208-431-2

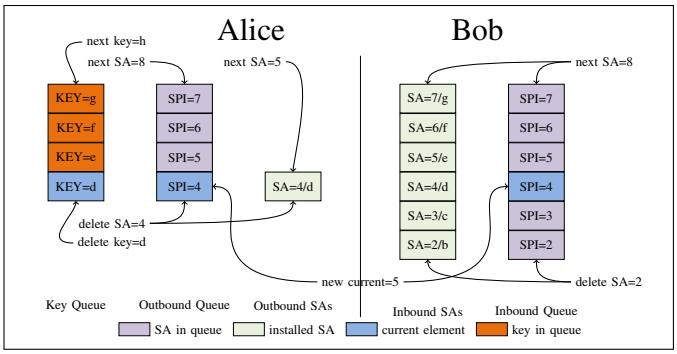

Figure 2. Key Change Process

key synchronization in case of lost and too early or too late arriving key change messages. The buffer of previously created SAs compensates desynchronization. For every receiver is able to calculate the according SPIs beforehand, it may, by comparing a received SPI with an expected, detect and correct the discrepancy by calculating the following SAs. Through this compensation process, there is neither need to interfere with the data communication nor to even inform the sender of lost key change messages; the sender may unperturbedly continue with data and control communications. This mechanisms make constant acknowledgements expendable and contribute thereby to a better protocol performance through omission of the round trip times for the majority of the necessary control messages. Because of this, acknowledgement messages (key change acknowledge) are still sent, but serve merely as a keepalive mechanism instead of true acknowledgements. In rare occasions, a key change message might be actually received, but the slave might not be able to apply the key for some reason (for instance issues regarding the QKD system or the Kernel). In this case, it reports the failure to the master with an appropriate message (key change fail). In case too many control packets go missing (what the receiver is able to detect by SPI comparisons and the sender by the absence of keepalive packets) or the key application fails, every peer is able to initiate a reset procedure (master or slave reset). The actual threshold of allowed and compensated missing messages is a matter of configuration and corresponds to the queue sizes for the SAs and therefore the ability of the system to compensate these losses. The master does not need to report key change fails, for it is in control of the synchronization process and might just initiate a reset if it is unable to apply its key. An additional occasion for a reset is the beginning of a conversation. At that point, the master starts the key synchronization process with an initial reset. A reset consists of clearing and refilling all of the queues and installed SAs. For the same reason as for the data channel, the authentication key for the control channel changes periodically. Due to the relatively low transmission rates on the control channel the key period is much longer (the software's default is 3 seconds) than on the data channel. As, therefore, control channel key changes are comparatively rare and reset procedures should only occur in extreme situations, both types implement a three way handshake. This is, on the one hand, because of the low impact on the overall performance due to the rare occurences, on the other hand due to higher impact of faulty packets. The control channel, however, implements the same SA buffering method as the data channel (only with AH SAs).

IV. Implementation

The presented solution, called *QKDIPsec*, consists of three parts (see also Figure 3):

- key acquisition;
- key application;
- key synchronization;

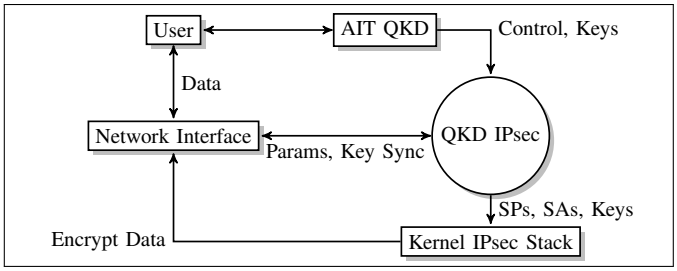

Figure 3. QKDIPsec Systems Context

Each of this tasks has a corresponding submodule inside QKDIPsec, while the overall control lies within the responsibility of the *ConnectionManager* class, which provides the main outside interface and instantiates the classes of said submodules using corresponding configuration. Also, all of these classes have corresponding configuration classes using a factory method pattern [13, p.134] and according configuration classes, decoupling program data and logic. The first task (key acquisition) is the objective of an interface to the AIT QKD software, the *KeyManager*, which provides the quantum key material. In this proof of concept, this class generates dummy key from a ring buffer, while it already has the according interfaces for the QKD software to serve as a class to acquire quantum key material and provide it in a an appropriate way to QKDIPsec. By now, only one function implementation is missing on the QKD software side to fully integrate QKDIPsec into the QKD software.

The second part (*KernelIPsecManager*) enters the acquired key directly into the Linux kernel, which encrypts the data sent to and decrypts the data received from a peer. Responsible for this part are a number of C++ classes, which control the SP and SA databases (SPD and SAD) within the Kernel's IPsec subsystem via the Linux *Netlink* protocol. Therefore, this solution uses the derived class *NetlinkIPsecManager*, but leaves the option to use other methods for kernel access as well. The reason for using Netlink to communicate with the kernel is that it was found the most intuitive of the available methods and that it is also able to handle not only the IPsec subsystem but a broad span of network functions in Linux. Furthermore, using a direct kernel API, as opposed to other IPsec implementations, omits middleware, both enhancing performance as well as eliminating potential source of error. Also using Netlink functions, this part governs the tunnel interfaces and routing table entries necessary for the communication via the classes *KernelNetworkManager* and *NetlinkNetworkManager* as well.

Netlink is a socket-oriented protocol and allows therefore the use of well-known functions from network programming.

 ISBN: 978-1-61208-431-2

The difference to the latter is that instead of network peers, communication runs within the system as *inter-process communication (IPC)*, through which also the kernel (via process ID zero) is addressable. Due to its network-oriented nature, a packet structure is used instead of function calls via parameters. This means that commands to the kernel (for instance to add a new SA) needs to be memory-aligned in the according packet structure and subsequently send to the kernel via a Netlink socket. A downside of Netlink during implementation was the complicated nature and weak documentation of its IPsec manipulation part (*NETLINK_XFRM*). While the Netlink protocol itself is present in every message in the form of its uniform header, the *NETLIN_XFRM* parts use a different structure plus individual extra payload attributes for every type of message (add and delete messages for both SAs and SPs), making the according class hierarchy rather inflated.

The key synchronization, eventually, is the main task of the *Rapid Rekeying Protocol*. As this is the very core of the solution, its implementation resides directly inside the connection manager. While it uses the classes mentioned above to acquire and apply the QKD keys in the manner discussed in Section III, it handles the key synchronization using sender and receiver threads (representing the master and slave parts, respectively), as well as a class for key synchronization messages. Within this class, also the described lost message compensation and reset, as well as initialization and clean-up procedures are implemented. The reset procedure may also include some re-initialization process for the QKD system, triggered via the *KeyManager*. This class also sets the clocking for the key changes, which is dynamically adjustable during runtime.

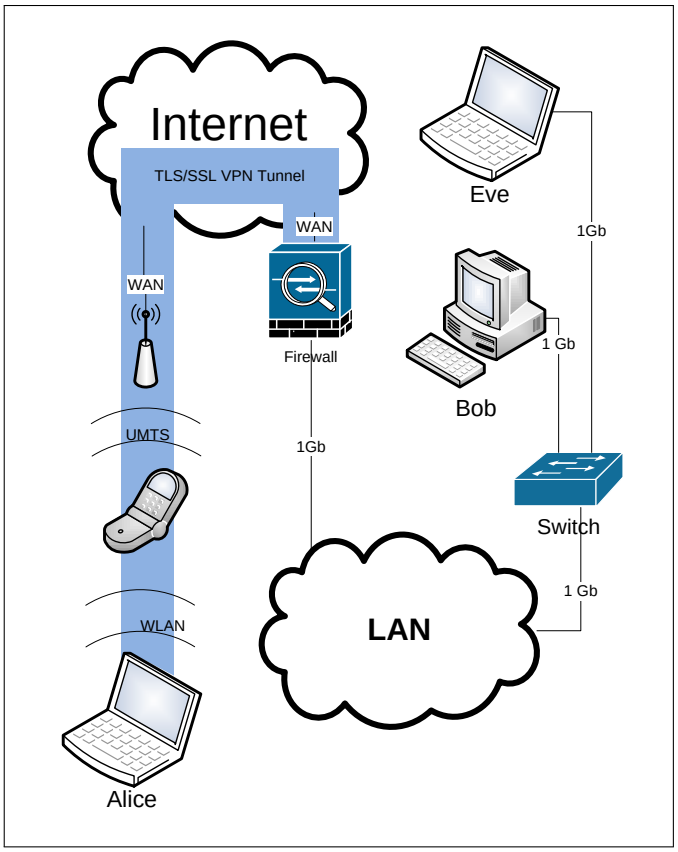

Figure 4. WAN Test Setup

V. Conclusion

The protocol design of the described solution aims on the one hand on speed and flexibility and on the other hand on fault tolerance, hence the architecture is as simple and lightweight as possible (including abandoning the IKE protocol). Due to this, very high IPsec key change rates can be achieved, even under harsh conditions. The solution was implemented in software using C++ and tested on two to five year-old Linux computers (Alice and Bob), both in a gigabit Local Area Network (LAN) and a UMTS-Wide Area Network (WAN) environment (the latter further aggravated by combining it with WLAN and an additional TLS-based VPN tunnel - see Figure 4) by means of data transfer time measurement and ping tests, as well as validation of the actual key changes by a Wireshark network sniffer (Eve).

Table I shows the results in seconds (four trials each, separated by slashes) of data transmission and in percent on ping tests within the mentioned LAN and WAN environments with various configurations: unencrypted, standard IPsec and QKDIPsec with different encryption algorithms, the latter also with different key periods. In these tests, both data transfer and ping were initiated by one peer (*Alice*). While the ping test was continuous, the data transfer consisted each of one data transfer from *Alice* to *Bob* and vice versa. The test file used on the LAN was a video file of 69.533.696 bytes size, while the WAN file was also a video, but only 1.813.904 bytes big. In both cases, key periods of 25 ms and less could be achieved, maintaining a stable data connection. This, using the recommended key length of 256 bit, surpasses the goal of 12,500 key bits per second (the currently maximal quantum key distribution rate under ideal circumstances), even though (deliberately) legacy equipment and a less-than-ideal network environment was used. Comparison of the performance shows a (expectable) higher data transfer period of QKDIPsec and unencrypted traffic, but no significant difference to traditional IPsec. Only the packet losses on a simultaneously running ping test were a few percentage points higher (mainly in the WAN environment).

TABLE I. Performance Test Results

LAN			
Setting	A→B	B→A	Ping
unencrypted	6/6/7/6	7/9/7/8	100%
AES-256 CCM			
standard IPsec	14/14/16/15	17/18/26/18	100%
50 ms	8/10/8/9	14/16/16/16	100%
25 ms	10/9/8/8	14/15/17/16	100%
20 ms	9/9/9/9	11/16/17/12	100%
AES-256 CBC			
20 ms	9/7/7	11/13/17	100%
Blowfish-448			
20 ms	14/9/7	15/13/14	99%
WAN			
Setting	A→B	B→A	Ping
unencrypted	10/10/10/10	9/7/6/7	99%
AES-256 CCM			
standard IPsec	11/11/11/11	11/5/6/5	99%
50 ms	14/10/11/13	6/5/5/5	95%
25 ms	10/11/10/10	6/7/6/7	94%
20 ms	12/11/13/10	9/5/6/6	98%
AES-256 CBC			
20 ms	10/11/11	9/7/8	100%

 ISBN: 978-1-61208-431-2

To verify the key changes, a network sniffer, Eve, was keeping track of the actual SPI changes of the packets transmitted between Alice and Bob. Table II shows a random sample of key change periods in milliseconds during the above mentioned LAN 20ms AES-256-CCM test. Within this table, the first column shows the key change times for data (ESP) packets from Alice to Bob while the second shows the opposite direction. As the recorded data contains one file copy from Alice to Bob (in the first half of the record) and one vice versa (in the second half), one randomly chosen sample of five consecutive key changes for each direction and from each half is chosen. This form of sample choosing from different phases and directions of the communication session and averaging them compensates inaccuracies, induced by the pause between key change and respective next following packet, which become greater the less traffic is sent. As the receiver only acknowledges received data and, therefore, sends significantly less packets, the vagueness of the non-averaged results is greater when receiving. The total average of all four of these averaged values is 0.020495 ms, which is approximately 2.5% above 20 ms per key change. This may be explained by the send and receive overhead for processing the key change messages, for the period is determines only the sleeping duration of a sender thread.

Because of the lower amount of traffic (due to the lower speed) and higher latency such exact time readings are not possible in the WAN environment. Therefore, the measurement method was changed to averaging a sample set of 20 key change periods, using the same random choosing as above. With approximately 0.2475, the total averaged result lies significantly higher (approximately 19%) than the one of the LAN setting. One possible explanation for this behavior is the latency in this environment.

	A→B		B→A	
	1st	2nd	1st	2nd
LAN	0.0220	0.0216	0.0208	0.0203
	0.0187	0.0204	0.0197	0.0235
	0.0145	0.0216	0.0203	0.0176
	0.0195	0.0243	0.0204	0.0197
	0.0225	0.0180	0.0207	0.0238
Ø	0.0194	0.0212	0.0204	0.0210
WAN $\sum 20$	0.5201	0.4899	0.4302	0.5397
Ø	0.0260	0.0245	0.0215	0.0270

TABLE II. Network Sniffing Results

Additionally, the recovery behavior was tested by letting the master deliberately omit key change notifications through the introduction of *if* clauses within the sending routine, while again running ping tests and file copies. Omitting single key change messages (and, thus, testing the recovery mechanism) yield in no measurable impact on the connection (along with 100% of successful pings). Also, by the same method of omitting key change requests, but this time surpassing the recovery queue size, the reset procedure was tested. The queue size was set to 50 and *Alice* was programmed to omit 50 sending key change messages after 200 sent ones. Expectedly, *Bob* initiated a reset procedure during the hiatus, resulting in a cycle of 200 key changes and a subsequent reset. Despite these permanent reset-induced interruptions, bidirectional ping tests only yielded insignificant losses (99.74% from *Alice* to *Bob* and 99.36% vice versa). Furthermore, a file copy in both directions was still possible.

These proof of concept tests show that using IPsec with appropriate key management is able to overcome the bandwidth restrictions of QKD, even when operating the data channels in less-than-ideal conditions. Furthermore, this paper presents an approach to provide QKD-secured links with high speeds meeting the bounds discussed in Section II, including a suitable performant and fault-tolerant key synchronization protocol (the *rapid rekeying protocol*) and a corresponding software solution running under Linux (*QKDIPsec*), that is to be integrated as a module into the AIT QKD software.

REFERENCES

[1] H. Zbinden, H. Bechmann-Pasquinucci, N. Gisin, and G. Ribordy, "Quantum cryptography," Applied Physics B, vol. 67, no. 6, 1998, pp. 743–748.

[2] M. A. Nielsen and I. L. Chuang, Quantum Computation and Quantum Information, ser. Lecture Notes in Physics. Cambridge: Cambridge University Press, 2000.

[3] A. Treiber et al., "A fully automated entaglement-based quantum cryptography system for telecom fiber networks," New Journal of Physics, no. 11, April 2009, p. 045013.

[4] F. Xu et al., "Field experiment on a robust hierarchical metropolitan quantum cryptography network," Chinese Science Bulletin, vol. 54, no. 17, 2009, pp. 2991–2997.

[5] S. Kent and K. Seo, "RFC4301: Security Architecture for the Internet Protocol," 2005.

[6] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for Key Management Part 1: General (Revision 3 - NIST Special Publication 800-57)," 2012, retrieved at July 10, 2015. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf

[7] A. Poppe et al., "Practical quantum key distribution with polarization entangled photons," Optics Express, vol. 12, no. 16, 2004, pp. 3865–3871.

[8] Internet Assigned Numbers Authority, "IPSEC ESP Transform Identifiers," 2012, retrieved at July 10, 2015. [Online]. Available: http://www.iana.org/assignments/isakmp-registry/isakmp-registry.xhtml#isakmp-registry-9

[9] J. Kang, K. Jeong, J. Sung, S. Hong, and K. Lee, "Collision Attacks on AES-192/256, Crypton-192/256, mCrypton-96/128, and Anubis," Journal of Applied Mathematics, vol. 2013, 2013, p. 713673.

[10] P. Hoffman, "RFC4308: Cryptographic Suites for IPsec," 2005.

[11] D. A. McGrew, "Impossible plaintext cryptanalysis and probable-plaintext collision attacks of 64-bit block cipher modes." IACR Cryptology ePrint Archive, vol. 2012, 2012, p. 623.

[12] "ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012)," 2012, retrieved at July 10, 2015. [Online]. Available: http://www.ecrypt.eu.org/ecrypt2/documents/D.SPA.20.pdf

[13] E. Freeman, E. Robson, B. Bates, and K. Sierra, Head First Design Patterns. Sebastopol: O'Reilly, 2004.

 ISBN: 978-1-61208-431-2