Classical Security Protocols for QKD Systems

Nikolaos Papanikolaou

Rajagopal Nagarajan

Department of Computer Science, The University of Warwick

Abstract

The purpose of this report is to document the three principal classes of classic cryptographic protocols which are needed in systems for quantum key distribution. We will detail the protocols used for secret–key reconciliation by public discussion, privacy amplification by public discussion, and unconditionally secure authentication. We suggest the use of the model checking method for the analysis of these protocols and the verification of relevant security requirements; the model checking technique is described, and the PRISM model checker is presented.

Key words: quantum cryptography – secret-key reconciliation – privacy amplification – authentication – unconditional security – model checking

1 Introduction

The term *quantum key distribution* (QKD) refers to a set of procedures — which involve the preparation, manipulation and measurement of quantum states— for establishing a common secret key between two or more users; for the purposes of discussion, we will restrict ourselves to the case of two users, Alice (the initiator) and Bob (the receiver or responder).

A number of cryptographic protocols implementing quantum key distribution have been proposed in the literature, among them BB84 [12], B92 [14] and E91 [19]. The use of quantum phenomena for key establishment provides benefits which are otherwise unachievable, namely:

- a communications channel that guarantees a certain degree of privacy;
- the detection of any attacker with an arbitrarily high probability.

The basic structure of a QKD protocol is roughly as follows. Firstly, Alice prepares a finite set of quantum particles in a known physical state; she then applies a particular transformation of her choice to each, and sends the particles to Bob over a quantum channel. Not knowing which transformation Alice has applied to each particle, Bob performs a quantum measurement of each and records the outcome. Then Alice and Bob discuss whether his measurements are *compatible* with her chosen transformations, and any discrepancies between the two are located and discarded. After this stage of the protocol, Alice and Bob share a common bit string, known as the *raw key.* Under ideal conditions and in the absence of an attacker, this is the same as the final secret key.

However, additional procedures normally need to be performed in order to provide a high level of security (i.e. in order to ensure the privacy of the key). Not only does a basic QKD protocol with the above structure fail to produce a secret key in the presence of an attacker, but it fails to provide a unique, common key in the presence of noise on the communications channel. Furthermore, it does not differentiate between legitimate users and the attacker. In order to address these issues, three techniques must be used:

authentication, a technique for ascertaining the identity of a particular user, so that an impersonation attempt is prevented;

reconciliation, a technique for detecting and correcting any discrepancies between Alice's and Bob's bit strings by disclosing a minimal amount of information to an attacker;

privacy amplification, a technique for distilling the final secret key from a longer, less secure bit string shared by Alice and Bob.

It has been shown [11] that the BB84 protocol [12], if supplemented by reconciliation and privacy amplification procedures, is unconditionally secure (i.e. the final key is guaranteed to be secret) against all attacks permitted by the laws of quantum mechanics. Furthermore, an authentication scheme is known [13] which is also unconditionally secure in the information-theoretic sense.

It should be noted that the techniques currently known for reconciliation [18], privacy amplification [16], and unconditionally secure authentication [13] do not involve the exploitation of any quantum mechanical phenomena and thus are referred to as "classical."

Classical communication protocols and security protocols have always been regarded as ideal targets for *automated verification,* i.e. for analysis by computer. Automated verification techniques include *model checking* and *theorem proving*, and are theoretically founded on sound mathematical principles, such as those of the theory of automata. The former of the two, which is of most interest to us here, involves an exhaustive search of all possible scenarios that

may arise in practice, including possible attack strategies. This leads to the efficient detection of protocol flaws, and contributes in a direct way to ensuring protocol correctness and suitability for particular applications, even before the protocols are actually implemented in practice.

Well–known tools for automated verification of classical *security* protocols include FDR [5], Murϕ [6], Brutus [7] and the NRL Protocol Analyzer [8]. Theoretical approaches to this problem include BAN Logic [9], Strand Spaces [10], and the inductive approach [2] associated with the HOL theorem prover. A widely known result in the area of automated verification of classical protocols is the discovery, by Gavin Lowe [3], of a subtle flaw in the Needham–Schroeder Public Key protocol using the FDR model checker.

1.1 Roadmap

This report will detail some of the most important classical protocols needed in the design of any practical QKD system, and it will present directions for research in the formal verification of these protocols. We will discuss the **Shell** and **Cascade** protocols for secret-key reconciliation by public discussion [18], the privacy amplification techniques discussed in [16], and the Wegman–Carter scheme for unconditionally secure authentication. These will be dealt with in sections 2, 3 and 4 respectively.

Section 5.1 discusses the model checking technique and presents the probabilistic model checking tool PRISM; this section is based on [32].

1.2 Setup

We assume in the following discussion that Alice and Bob are able to communicate through two channels with different properties, namely a quantum channel for performing QKD and an authenticated public channel. Formally, the setup involves the two users, Alice and Bob, an attacker Eve, and two channels between Alice and Bob with the following specifications:

(1) a "classical" channel, with perfect authenticity but no privacy;
(2) a "quantum" channel, with imperfect privacy but perfect authenticity.

The objective of Alice and Bob, after having performed a basic QKD protocol on the quantum channel obtaining respective bit strings A and B of length n, is to establish a shorter, common secret string S. After reconciliation of strings A and B is performed, a common string T is produced, of length n. The process of privacy amplification discards some of the bits in T, resulting in

S. Both reconciliation and privacy amplification are performed on the classical channel. In both cases, the essential goal is to minimize (and when possible eliminate completely) the amount of correct information Eve has about T and S.

2 Secret–Key Reconciliation Protocols

The reconciliation problem is to design a protocol R^P that runs on strings A and B to produce a secret string S, while exchanging a minimal quantity Q of information regarding S over the public channel. We write:

$$R^P(A, B) = [S, Q]$$

to express the function of protocol R^P, and we denote the average amount of information leaked to an attacker during the protocol by $I_E(S|Q)$. The reconciliation problem forms the object of L. Salvail's M.Sc. thesis [20] and is also presented in [18]; in both of these sources, the emphasis is put on the design of efficient and implementable reconciliation protocols.

2.1 Components of Reconciliation Protocols

Reconciliation protocols are made up of particular building blocks, also known as *primitives*. Primitives usually operate on binary strings (i.e. strings $w \in \{0,1\}^n$ for particular n), typically Alice's and Bob's strings A, B or substrings of these. The most common primitives and their interrelationships are presented in this section. The primitives discussed include the computation of the parity of a given string, an interactive binary search procedure, a procedure for determining whether two strings match, and more. The terminology used here, and the naming conventions, are based on [20]. The primitives **PARITE, DICHOT, DIPAR, CONFIRME, DICONF** will be detailed in turn. Before proceeding to consider the reconciliation protocols **Shell** (originally known as **Coquille**, see [20]) and **Cascade** [18], a typical so-called *protocol kernel* will be presented.

2.1.1 The **PARITE** *Primitive*

PARITE is a primitive which Alice and Bob use in order to determine whether any corresponding substrings of A and B have equal parity (parity is even, i.e. has value 0, when there is an even number of 1s in a string; similarly, it is odd and had value 1 if there is an odd number of 1s in a string).

A_i and B_i denote the i–th bit in Alice's and Bob's sequence, respectively; the symbol $\oplus$ denotes bitwise exclusive disjunction (the XOR operation). The **PARITE** primitive is as follows.

(1) Alice sends to Bob the value:

$$p := \bigoplus_{i=1}^{n} A_i$$

(2) Bob privately computes the value:

$$p' := \bigoplus_{i=1}^{n} B_i$$

(3) Bob computes $(p \oplus p')$ to determine whether there is an odd or even number of errors in his received sequence (B) and communicates this to Alice.

Example 1 *Suppose* $A = \{1, 0, 1, 1, 1\}$ *and* $B = \{1, 1, 1, 1, 1\}$. *Running* **PARITE**(A, B) *gives* $p = 0$, $p' = 1$, *and* $p \oplus p' = 1$. *Hence there is an odd number of errors in* B, *in this example evidently only one.*

2.1.2 The **DICHOT** *Primitive*

DICHOT is a distributed variation of the binary search algorithm, in which Alice and Bob recursively divide A and B and compare the parities of corresponding blocks until a discrepancy is found. This allows Alice and Bob to locate an error in B without divulging the values of individual bits in the string (except for the erroneous one). DICHOT only succeeds if there is an *odd number of errors*. The steps of **DICHOT** are as follows.

(1) If $|A| = 1$, then Bob requests the value of A explicitly and Alice sends it to him. Then the primitive completes successfully.
(2) Otherwise, Alice privately splits A into two halves:

$$\text{left half}: \ \hat{A} = \{A_1, A_2, \ldots, A_{\lfloor |A|/2 \rfloor}\}$$
$$\text{right half}: \ \hat{A}' = \{A_{\lfloor |A|/2+1 \rfloor}, A_{\lfloor |A|/2+2 \rfloor}, \ldots, A_{|A|}\}$$

(3) Similarly, Bob privately splits B into:

$$\text{left half}: \ \hat{B} = \{B_1, B_2, \ldots, B_{\lfloor |B|/2 \rfloor}\}$$
$$\text{right half}: \ \hat{B}' = \{B_{\lfloor |B|/2+1 \rfloor}, B_{\lfloor |B|/2+2 \rfloor}, \ldots, B_{|B|}\}$$

(4) Alice and Bob perform **PARITE**$(\hat{A}, \hat{B})$.

(5) If $(p_A \neq p_B)$, then Alice and Bob perform **DICHOT**$(\hat{A}, \hat{B})$ (recursive call with left halves).
(6) If $(p_A = p_B)$, then Alice and Bob perform **DICHOT**$(\hat{A}', \hat{B}')$ (recursive call with right halves).

Example 2 *Suppose* $A = \{1, 0, 1, 1, 1\}$ *and* $B = \{1, 1, 1, 1, 1\}$. *Calling* **DICHOT**(A, B) *produces* $\hat{A} = \{1, 0\}$ *and* $\hat{B} = \{1, 1\}$, *for which* $p_A = 1$ *and* $p_B = 0$ *respectively. The recursive call* **DICHOT**$(\{1, 0\}, \{1, 1\})$ *is then made, setting* $\hat{A} = \hat{B} = \{1\}$ *and* $p_A = p_B = 1$, *so the next recursive call is made on the right halves of* $\{1, 0\}$ *and* $\{1, 1\}$, *i.e.* **DICHOT**$(\{0\}, \{1\})$. *Since the length of the arguments is now 1, the error in the second location of* B *has been found. In order to implement* **DICHOT** *in practice, the index (with respect to* A *and* B*) of the first element in* $\hat{A}$ *and* $\hat{B}$ *must be recorded, so that the actual index of the error can be returned.*

A related primitive, known as **DIPAR**, performs a parity check and calls DICHOT if the parities of its arguments do not match. We write:

$$\textbf{DIPAR}(A, B) := \text{If } \textbf{PARITE}(A, B) = \textbf{false} \text{ then } \textbf{DICHOT}(A, B)$$

DIPAR is used in the protocol **Cascade**, described later.

2.1.3 The **CONFIRME** *Primitive*

The **CONFIRME** primitive allows Bob to determine if his bit string, B, differs from A. **CONFIRME** succeeds in detecting a discrepancy between A and B (if there is indeed one) with probability $\frac{1}{2}$; however, if A and B are identical, **CONFIRME** is guaranteed to detect this. If **CONFIRME** is applied repeatedly, namely k times, to A and B, it has a probability of failure of $\frac{1}{2^k}$. **CONFIRME** consists of the following steps:

(1) Alice and Bob decide publicly (i.e. over the public channel) on a random subset of A and B by choosing a random bit string

$$\lambda \in \{0, 1\}^n$$

where $\lambda_i = 1$ indicates that the i–th bit of Alice's and Bob's sequences will be used in the following calculation.
(2) Alice and Bob compare the parities of the chosen subset of A and B (denoted by λA or $\lambda \cdot A$, and by λB or $\lambda \cdot B$ respectively, where $\cdot$ denotes bitwise conjuction). In other words, they perform **PARITE**$(\lambda \cdot A, \lambda \cdot B)$.
(3) If $(p' \neq p)$ then Bob knows that $A \neq B$. Otherwise he concludes $A = B$.

CONFIRME is part of the **DICONF**^{k} primitive, discussed next. An illustration of **CONFIRME** is shown in Example 3.

2.1.4 The **DICONF**^{k} *Primitive*

DICONF^k is a combination of **DICHOT** and **CONFIRME** which can correct up to k errors in Bob's received string. The idea of this primitive is to run **CONFIRME** repeatedly (k times) on A and B until a random subset of these containing an error is found; then this particular subset is searched using **DICHOT** until the index of the error is located, and this is then corrected. So, every time **CONFIRME** finds an error, that error is corrected and the whole procedure is applied to the corrected bit string. Importantly, **DICONF**^{k} only succeeds if there is an *odd number of errors* in Bob's string. The steps of **DICONF**^{k} are shown in detail below.

(1) If $(k = 1)$ then return B.
(2) $c :=$ **CONFIRME** (A, B).
(3) If $(c =$ **false**$)$ then:
  (a) $loc :=$ **DICHOT**$(\lambda A, \lambda B)$.
  (b) $B'_{loc} := B_{loc}$.
(4) Else if $(c =$ **true**$)$ then $B' := B$.
(5) **DICONF**$^{k-1}(A, B')$

Example 3 *Suppose* $A = \{0, 1, 0, 1, 1, 1\}$ *and* $B = \{0, 0, 0, 1, 0, 0\}$. *Three errors have occurred. We will run* **DICONF**$^1(A, B)$.

Let $\lambda = \{0, 0, 0, 0, 1, 0\}$ *be the random subset chosen by Alice and Bob when* **CONFIRME** *is run; in other words, the chosen subset consists of the second last bit of* A *and* B *respectively. The parities of the subsets are*

$$p = \bigoplus_{i=1}^{6} \lambda_i A_i = (0 \cdot 0) \oplus (0 \cdot 1) \oplus (0 \cdot 0) \oplus (0 \cdot 1) \oplus (1 \cdot 1) \oplus (0 \cdot 1) = 1$$

$$p' = \bigoplus_{i=1}^{6} \lambda_i B_i = (0 \cdot 0) \oplus (0 \cdot 0) \oplus (0 \cdot 0) \oplus (0 \cdot 1) \oplus (0 \cdot 1) \oplus (0 \cdot 0) = 0$$

Since $(p' \neq p)$, *Alice and Bob have detected that* $A \neq B$. *They now run* **BINARY** *and locate the discrepancy between* A_5 *and* B_5. *The error is corrected and the procedure finishes with* $B' = \{0, 0, 0, 1, 1, 0\}$.

2.1.5 Kernels for Reconciliation Protocols

A *kernel* for a reconciliation protocol is a minimum-distance decoding procedure that Bob performs on his received sequence in order to correct some errors. In [20], the complexity and optimality of such procedures is studied in detail. The following protocol kernel, which is of interest to us, has been

shown to be optimal for an adequate choice of the integer k. We will name it $\mathbf{KERNEL}_1$.

(1) Alice and Bob choose at random a function $f : \{0,1\}^n \to \{0,1\}^k$, where k is a parameter to be determined. The description of f is disclosed on the public channel.
(2) Alice sends Bob the value of $f(A)$.
(3) Bob computes a sequence B' with a minimum Hamming distance from B, such that $f(B') = f(A)$. He replaces B by B'.

The problem with $\mathbf{KERNEL}_1$ is that it involves a random choice between *all* functions from binary strings of length n to binary strings of length k (there are 2^{k2^n} of these). Assuming $k > nh(p)$, where $h(p)$ is the entropy of a Bernoulli trial with probability p, to represent such a function one needs to store $k2^n$ bits, and therefore the process of choosing and evaluating such a function is computationally expensive.

It turns out that we can impose a constraint on the class of functions from which f is chosen in $\mathbf{KERNEL}_1$, and still maintain optimality. The $\mathsf{universal}_2$ hash functions proposed by Wegman and Carter [13] can be used in this context. The operation of selecting at random a $\mathsf{universal}_2$ hash function is efficient, and these functions can be computed efficiently also. In practice, the kernels used in reconciliation protocols involve a random choice of f from the class of $\mathsf{universal}_2$ hash functions.

Definition 4 ($\mathsf{universal}_2$ functions) *Let H be a class of functions with domain A and co-domain B. The class H is said to be $\mathsf{universal}_2$ if, for every pair of distinct elements $x, y \in A$,*

$$\delta_H(x,y) \leqslant \frac{|H|}{|B|}$$

where $\delta_H(x,y)$ denotes the number of functions in H which map x and y to the same element in B.

The **Cascade** reconciliation protocol simply uses the **DIPAR** primitive as a kernel, according to [20, p.60].

2.2 The **Shell** *Reconciliation Protocol*

The **Shell** protocol combines the kernel described in section 2.1.5 with the $\mathbf{DICONF}^k$ primitive in order to reconcile Alice's and Bob's bit strings A and B. **Shell** starts by splitting Alice's and Bob's sequences into blocks, and then performs the kernel on each block. For each block, $\mathbf{DICONF}^\mathbf{1}$ is applied

repeatedly in order to locate and correct errors; then pairs of adjacent blocks are joined together, and the procedure is repeated on the resulting superblocks. The steps of **Shell** are detailed below. Each repetition of the procedure is referred to as a *pass*.

(1) Alice and Bob split their bit strings into disjoint blocks of length k, namely

$$\begin{aligned} \text{for Alice}&: A_0[1], A_0[2], \ldots, A_0[\lceil n/k \rceil] \\ \text{for Bob}&: B_0[1], B_0[2], \ldots, B_0[\lceil n/k \rceil] \end{aligned}$$

(2) Alice and Bob perform $\mathbf{KERNEL}_1(A_0[1], B_0[1])$, $\mathbf{KERNEL}_1(A_0[2], B_0[2])$, $\ldots$, $\mathbf{KERNEL}_1(A_0[\lceil n/k \rceil], B_0[\lceil n/k \rceil])$ in that order.

(3) $s := 0$. This variable is used as a subscript specifying the current pass. The following is repeated $(s+1)$ times.

(a) For $1 \leqslant i \leqslant \left\lceil \frac{n}{2^s k} \right\rceil$ repeat the following:

(i) Alice and Bob perform $\mathbf{DICONF^1}(A_s[i], B_s[i])$.

(ii) If Bob locates an error, he asks Alice for the contents of the whole block that contains the error. He replaces his corresponding block with the block sent by Alice.

(b) If $\left\lceil \frac{n}{2^s k} \right\rceil > 1$ do the following:

(i) $s := s + 1$.

(ii) Alice and Bob concatenate the block left of the current one with the block to the right of the current one and this forms the block used in the next pass ($\circ$ denotes concatenation of binary strings):

$$\begin{aligned} A_s[i] &:= A_{s-1}[2i-1] \circ A_{s-1}[2i] \\ B_s[i] &:= B_{s-1}[2i-1] \circ B_{s-1}[2i] \end{aligned}$$

The **Shell** protocol is optimal, in the sense that the average amount of information gained by an attacker (his *equivocation*) is minimal. However **Shell** has a substantial running time and is not efficient enough to be used for practical purposes. The **Cascade** protocol, described next, is more efficient.

2.3 The **Cascade** *Reconciliation Protocol*

Reconciliation protocols and the primitives they involve make extensive use of parity checks; thus, some of the primitives for detecting and correcting errors can only succeed if there is an *odd* number of errors in the pairs of strings or substrings to which they are applied. It follows that an efficient reconciliation protocol is one which splits Alice's and Bob's strings into substrings with an odd number of errors as frequently as possible. In order to reduce the number of misses, a protocol should vary the block size used at each pass and perform parity calculations repeatedly. The **Shell** protocol has a fixed block size k,

which is established at the beginning of the protocol; on each pass, the size of the block is doubled, since adjacent blocks from pass s are joined together to form the block in pass $s+1$. The **Cascade** protocol, on the other hand, is distinguished by the following two characteristics:

- varying block sizes at each pass;
- a random permutation is chosen and applied to the current block at each pass.

Below is a listing of the **Cascade** protocol.

(1) Alice sets $A_0 := A$.
(2) Bob sets $B_0 := B$.
(3) For $1 \leqslant i \leqslant \rho$ repeat the following:
    (a) Alice and Bob select at random a permutation σ_i of $\{1, 2, \ldots, n\}$ and exchange the details of this permutation over the public channel.
    (b) Alice computes the string $A_i := \sigma_i(A_{i-1})$.
    (c) Bob computes the string $B_i := \sigma_i(B_{i-1})$.
    (d) For $1 \leqslant j \leqslant \lceil n/k_1 \rceil$ repeat the following:
        (i) Alice and Bob run **DIPAR**$(A_i^{k_i}[j], B_i^{k_i}[j])$.
        (ii) If Bob corrects a single error in B, then Alice and Bob run procedure **CASCOR**, which is detailed in [20].

The **CASCOR** procedure starts with a block containing an odd number of errors and splits it repeatedly until all the errors have been corrected. The idea is to perform the splitting so that on each iteration of **CASCOR** there is an odd number of errors to correct, and the details of how this is performed are to be found in L. Salvail's M.Sc. thesis, cited above.

2.4 Summary

In this section we have considered several primitives for detecting and correcting errors in Bob's received bit string, assuming the scenario/setup established in Section 1.2. Most of the primitives detailed in section 2.1 are combined together to form other primitives (e.g. **DICONF**^{k} combines **BINARY** and **CONFIRM**, and both of the latter make use of **PARITE**). Reconciliation protocols make use of a *kernel*, which is a basic decoding procedure for correcting some errors in Bob's string. Kernels involve a random choice of a binary decoding function, and in practice the $\mathsf{universal}_2$ class of hash functions is used. The **Shell** protocol for reconciliation makes use of a kernel which selects among $\mathsf{universal}_2$ functions, and the protocol has been shown to be optimal but inefficient [20]. Finally, the **Cascade** protocol is more efficient than **Shell**, and is characterised by a varying choice of block size and the use of a random permutation.

3 Privacy Amplification Protocols

As discussed in section 1, the purpose of a privacy amplification protocol is to minimize, and if possible to eliminate completely, any information gained by the attacker, Eve, by eavesdropping on the classical and/or quantum channel. Note that the process of reconciliation leaks a certain amount of information to Eve about the common key T shared by Alice and Bob, since parities of various subsets of T are disclosed during this process, and information such as which bits are kept, and which are discarded, is disclosed on the classical channel. Remember that Eve is allowed to eavesdrop on the classical channel, but since this channel is authenticated and assumed to be perfect, Eve cannot tamper with any of the transmissions made thereupon. However, Eve is assumed to be able to tamper with the quantum channel, e.g. she is able to delete transmissions, to inject errors etc. In principle, it is quite possible for Eve to prevent communications between Alice and Bob through this channel completely; however this particular scenario is of little theoretical interest, because in this case any attempt at key establishment is impossible.

Privacy amplification comes in several flavours, depending on the assumptions made about the information available to Eve regarding Alice's and Bob's keys. In [15,16] three cases are considered:

(1) The case in which no eavesdropping has occurred on the classical channel during reconciliation, but tampering and transmission errors have occurred on the quantum channel ("public channel eavesdropping").
(2) The case in which a limited amount of eavesdropping has occurred on the classical channel (and has been detected), but a reconciliation protocol has *not* been performed (presumably because neither transmission errors nor tampering are assumed to have occurred). The fact that no reconciliation protocol has been performed at all deprives Eve of useful information regarding the key shared by Alice and Bob ("private channel eavesdropping").
(3) The case in which eavesdropping has occurred on the classical channel, and a reconciliation has been performed by Alice and Bob. This is the most general case ("public and private channel eavesdropping").

We consider only the first two cases here for simplicity, basing the discussion primarily on [17,15].

3.1 Reducing the Information of a Public Channel Eavesdropper

Let $f : \{0,1\}^n \to \{0,1\}^k$ be the decoding function used during reconciliation (we assume only $\mathbf{KERNEL_1}$ is executed) in this particular case. Eve's knowl-

edge includes the description of f and the value of $f(T)$. Thus Eve knows the set of all binary strings with length k which are similar to T (i.e. strings Z with a Hamming distance $\text{dist}(Z,T) \leqslant \ell$ for ℓ chosen by Eve):

$$C = \{Z \in \{0,1\}^n \mid f(Z) = f(T)\}$$

For Eve, all elements of C are equally probable candidates for T.

A protocol (which we will call $\mathbf{PA_1}$) for reducing Eve's information about T is given below. The security parameter s is a value between 0 and $(n-k)$.

(1) Alice chooses at random a function

$$g : \{0,1\}^n \to \{0,1\}^{n-k-s}$$

(2) Alice sends a description of g to Bob over the classical/public channel.
(3) Alice and Bob apply g to their (common) strings, so that the final secret key is $S = g(T)$.

This protocol produces a final key S of length $(n-k-s)$ such that the average information Eve has about S is less than $\log_2(1+2^{-s})$ bits (from [15]).

Protocol $\mathbf{PA_1}$ is too inefficient to be used in practice, since the description of g will require the transmission of up to $(n-k-s)2^n$ bits. In order to improve the efficiency of the procedure, g can be selected at random from a restricted class of functions. According to [15,16], a good choice is to use the class of *equitable functions,* defined as follows.

Definition 5 *If $i < j$, a function $f : \{0,1\}^j \to \{0,1\}^i$ is **equitable** if $|Y| = 2^{j-i}$, where $Y = \{x | f(x) = a\}$, for every binary string a of length i.*

Protocol $\mathbf{PA_2}$ is identical to $\mathbf{PA_1}$, except for the fact that the function g is selected at random from the class of equitable functions. The average information Eve has about the final key S in this case is the same as for $\mathbf{PA_1}$, i.e. $\log_2(1+2^{-s})$ bits. Examples of equitable functions that can be used in $\mathbf{PA_2}$ include $g_1(x) = x \operatorname{div} 2^{k+s}$ and $g_2(x) = x \bmod 2^{n-k-s}$, which are both computable in linear time.

A further variation on $\mathbf{PA_1}$ arises when we restrict the choice of g to the class of $\mathsf{universal}_2$ hash functions. An example of such a protocol [15], which we will call $\mathbf{PA_3}$, is shown below. This protocol assumes that reconciliation has been done using $\mathbf{KERNEL_1}$, with the decoding function $f_{a,b}(x) = (ax+b) \bmod p \bmod 2^k$. The class of functions $\{f_{a,b} | a, b \in \mathbb{Z}_p \text{ and } a \neq 0\}$ is $\mathsf{universal}_2$ [13,15].

(1) Alice fixes the function

$$g_{a,b}(x) = (ax+b) \bmod p \operatorname{div} 2^k$$

where a, b are the same values that were used in the decoding function for reconciliation.

(2) Alice sends a description of $g_{a,b}$ to Bob over the classical/public channel.

(3) Alice and Bob apply g to their (common) strings, so that the final secret key is $S = g_{a,b}(T)$.

The average information gained by Eve, if $\mathbf{PA_3}$ is used, is bounded above by the value of 2 bits (for a proof, see [15, p.40]).

3.2 Reducing the Information of a Private Channel Eavesdropper

If Eve eavesdrops on the private/quantum channel, and no reconciliation is performed at all, then Eve gains a certain amount of information on the key T established by Alice and Bob, but is deprived of highly valuable information regarding T that would have been disclosed over the public channel. In this case, the information about T available to Eve consists of the set

$$E = \{Z \in \{0,1\}^n \,|\, e(Z) = e(T)\}$$

where $e : \{0,1\}^n \to \{0,1\}^k$ is a function chosen randomly by Eve (and unknown to Alice and Bob). This function e represents the k–bit information Eve has about T. Alice and Bob only have a previously agreed upper bound on k. Since Alice and Bob do not have complete knowledge of E, it is not possible for them to eliminate Eve's information with certainty.

Alice and Bob need to agree on some function $g : \{0,1\}^n \to \{0,1\}^r$, for some $r \leqslant n - k$, so that knowledge of $e, e(x)$, and g leaves Eve with an arbitrarily small fraction of 1 bit of information about $g(x)$. Differing in the manner in which the function g is chosen, two privacy amplification protocols are available in this case – let's call them $\mathbf{PA_4}$ and $\mathbf{PA_5}$. In $\mathbf{PA_4}$, g is chosen uniformly at random from the set of all functions $\{0,1\}^n \to \{0,1\}^r$. For this protocol, the expected amount of information on $g(x)$ gained by knowledge of $e, g, e(x)$ where $e : \{0,1\}^n \to \{0,1\}^k$, $s < n - k$, $r = (n - k - s)$ is at most $\frac{2^{-s}}{\ln 2}$bits.

In protocol $\mathbf{PA_5}$, the function $g(x)$ is selected from the class of strongly universal$_2$ hash functions [13,15]. The expected amount of information gained by Eve on $g(x)$ is the same as before, i.e. at most $\frac{2^{-s}}{\ln 2}$bits.

4 Protocols for Unconditionally Secure Authentication

The protocols for secret-key reconciliation and privacy amplification discussed in the previous sections assume the existence of an authenticated public channel linking Alice and Bob. The purpose of this section is to briefly describe an unconditionally secure technique for providing that authentication capability. An authentication procedure must be applied before a QKD protocol is performed, and hence well before reconciliation and privacy amplification protocols.

Alice and Bob authenticate by sending each other a fixed message $M \in \{0,1\}^q$ with an authentication tag $t = v(M)$, where v is a function that they have agreed on. In order to agree on the function v, they need to share a short, common bit string w. We assume that w is established once, when Alice and Bob meet in person prior to the key exchange procedure. Wegman and Carter [13] have proposed that the function used to compute the authentication tag should be chosen at random from the class of strongly universal$_2$ hash functions. However, if v is chosen at random from this class of functions, the length of the message M becomes unnecessarily large. Unconditionally secure authentication can equally well be provided using a function v which is ε–almost strongly universal$_2$. This class of functions is defined as follows.

Definition 6 *Let M and J be finite sets, and call the functions from M to J "hash functions." Let ε be a positive real number. A set H of hash functions is ε–almost strongly universal$_2$ if the following two conditions are satisfied:*

(1) The number of hash functions in H that takes an arbitrary $m_1 \in M$ to an arbitrary $j_1 \in J$ is exactly $\frac{|H|}{|J|}$.
(2) The fraction of those hash functions that also takes an arbitrary $m_2 \neq m_1$ in M to an arbitrary $j_2 \in J$ (possibly equal to j_1) is no more than ε.

Example 7 *An example of a 1–almost strongly universal$_2$ family of functions is the $|J|$ hash functions defined by $h_i(m) = (m + i) \bmod |J|$.*

Let H be an ε–almost strongly universal$_2$ family of hash functions. Assume that Alice and Bob share a common bit string w just large enough to be able to agree on a hash function $h_k \in H$, $0 \leqslant k < |H|$. In order for Alice to authenticate herself to Bob:

(1) Alice sends Bob a message m_1 and the tag $t = h_k(m_1)$.
(2) Bob receives m_1 and a (potentially tampered) tag t'.
(3) Bob then computes $h_k(m_1)$ and compares it with t'. If they match, Bob accepts the message as authentic, i.e. as originating from Alice.

This protocol, and variations thereof, can be used to provide an authentication

capability in a QKD system [4].

5 Techniques for Analysing Security Protocols for QKD

5.1 Automated Verification by Model Checking

The amount of intelligence a computing machine can demonstrate has always been hotly debated; however, computers nowadays do have limited ability in assisting human reasoning and constructing mathematical proofs automatically. This is the result of many years' development of formal, mechanical techniques for analysing system behaviour. Indeed, the study of *automated verification,* as it is known, is an important part of any computer scientist's training [21]. Automated verification techniques include *theorem proving* and *model checking.*

On the one hand, theorem proving tools provide mechanical assistance in developing logical proofs. To show the validity of a given statement, a *theorem prover* aids the user in applying the inference rules of a particular logic, and maintains a history of the steps taken.

Model checking, on the other hand, is a procedure involving three main steps: constructing an abstract model of a given system *(system specification);* defining the properties desired of the system in a form that can be checked automatically *(property specification);* and feeding the model into an appropriate software tool *(verification).* A *model checker* then employs its built–in algorithms to prove, with little or no user intervention, whether the system model satisfies the properties given.

The latter of these two approaches to verification has been used in our work to investigate the properties of certain quantum protocols. In particular, we have used *logical* model checking [22] and *probabilistic* model checking [23] to assess the BB84 protocol [12] for quantum key distribution. For details, see [32,34].

Firstly, we will discuss, in turn, the three phases of model checking. This includes an inquiry into the syntax of description languages and temporal logic.

5.1.1 System Specification, and Description Languages

System specification is arguably the most crucial step when performing model checking for a particular problem. The system to be analysed has to be de-

scribed accurately in some general–purpose specification language; the description, or *model,* must incorporate all the salient features of the system's behaviour, and particularly those aspects of the system relevant to verification.

For a general communications protocol, there are several levels of abstraction at which a description can be made; frequently in protocol verification the emphasis is put on concurrency aspects and timing. It is of utmost importance that all users of a protocol interact in the correct order, and that data is not lost due to synchronisation errors. At this level of abstraction, however, it is immaterial what data representation is used, or whether a particular compression algorithm is involved. A suitable protocol model for analysing timing and other concurrency–related issues will abstract away from low–level considerations such as those just mentioned.

The situation is similar, but certainly more complex, in the analysis of *security* protocols. A suitable model in this case must take into account the details of encrypting and decrypting procedures, the availability and secrecy of keys, and the nature of the communication channels used. A specification language for security protocols will necessarily be more expressive than one intended for the protocols of the previous paragraph.

The use of process calculi as specification languages for protocols is quite common. Robin Milner's CCS, which was developed on similar lines as CSP, evolved into the π–calculus [25] and is also well–suited for this task. Interestingly, [26] extended the π–calculus with cryptographic primitives and other features relevant to security protocols; the result is the so-called spi–calculus . In more recent work, Gay and Nagarajan used CCS to model the BB84 quantum cryptographic protocol [27] and demonstrated the inability of an eavesdropper to succeed for a certain kind of attack. They subsequently developed a quantum process algebra, CQP, especially for the definition of quantum protocols [28].

5.1.2 Property Specification

Thus far, we have only considered means for describing system behaviour. However, this is insufficient for model checking, whose purpose is to demonstrate conclusively that a system operates in a desirable manner, and that it is free from design faults. Expressing precisely what features of a system are 'desirable' and exactly what constitutes a fault are the objectives of *property specification.* A *property* is any pattern of observable behaviour that a system should or should not exhibit; the function of a model checker is, thus, to show whether a system *satisfies* a given set of properties.

A property can be expressed as a *formula* in a given *logic.* Typically a system model is represented by a finite or infinite *automaton,* and the model checker

must determine the truth or falsity of the statement

$$\sigma \models \Phi \tag{1}$$

which means [22], "the run σ of the automaton representing the system model *satisfies* formula Φ".

Properties of a system model are usually expressed as formulae in temporal logic; in some model checking systems however, the behaviour defined by a property is described explicitly instead. For example, the SPIN model checker converts properties written in LTL (Linear Temporal Logic) to patterns of behaviour ("never claims") expressed in PROMELA. The PRISM model checker requires that properties are expressed in PCTL.

5.1.2.1 Temporal Logic Temporal logic was first recommended by [30] as a tool for reasoning about program computations. While propositional logic allows one to make statements about Boolean variables using various connectives, temporal logic includes *modal operators* that quantify such statements over time.

If a program computation is regarded as a sequence of states

$$s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n \quad \text{or, put otherwise,} \quad s_0 \xrightarrow{*} s_n$$

then one such state s_i may be regarded as the "present" moment in time, and all subsequent states are then moments in the future. The modal operators of temporal logic are used to quantify over present and future states.

Modal operators are applied to logical propositions; logical propositions consist of *atomic* propositions (which are regarded as uninterpreted symbols) and connectives such as $\neg$ ("not"), $\wedge$ ("and"), $\vee$ ("or") and $\Rightarrow$ ("implies").

The most commonly used operators in temporal logic are $\Box$ ("henceforth"), $\Diamond$ ("eventually"), and $\circ$ ("next"). The formula $\Box P$ ("henceforth, P") means that P is true for all states in a computation. For a computation whose present state is s_i and whose "future" are all states s_j $(j > i)$, the formula $\Box P$ states that the proposition P is true in state s_i and will remain true for all s_j.

The formula $\Diamond P$ ("eventually, P") states that there is some point in the computation at which P is true. If s_i is the present state of a computation, then $\Diamond P$ means that, either P is true in s_i or it will be true at some time in the future.

Finally, $\circ P$ means that P is true in the *second* state of a computation (i.e. in state s_{i+1} if s_i is the present state). The three modal operators can be combined together to express more complex properties. Any unbroken sequence of operators is termed a *modality,* and the number of operators in the sequence is the *degree* of that modality.

5.1.2.2 Linear Temporal Logic (LTL) versus Computation Tree Logic (CTL) There are two possible views regarding the nature of time, and each of these gives rise to a different class of temporal logic. In mainstream model checkers, two kinds of temporal logic are actually used: *linear* and *branching* temporal logic [31].

Linear temporal logic (LTL) treats time in such a way that, each moment has a *unique* possible future. Thus any LTL formula is interpreted over linear sequence, and essentially describes the behaviour of a single program computation. LTL is used, for instance, in the model checker SPIN.

On the other hand, in a *branching* temporal logic (or *computation tree* logic, CTL), each moment in time may have several possible 'futures'. Therefore, formulae in such a logic are represented by infinite computation trees; each tree describes a possible computation of a non–deterministic program. The PRISM tool uses a branching temporal logic, known as PCTL (*probabilistic* computation tree logic). PCTL caters for probabilistic computations, and the trees representing a given formula are labelled with probabilities.

5.1.3 Verification

The final phase of a model checking solution to a given problem involves *applying an automated tool* to the system description and the specification of its properties. In the literature, properties that express desired system behaviour are termed *liveness* properties, while *safety* properties express the absence of undesirable system behaviour [22]. Clearly, the model checker is expected to prove that liveness and safety properties do hold for a given system.

5.1.4 Probabilistic Model–Checking

PRISM is an acronym for *probabilistic symbolic model checker,* and is designed for modelling and validating systems which exhibit probabilistic behaviour. Whereas a logical model–checker, such as SPIN [22], only states whether a system model σ satisfies a temporal formula Φ, a tool such as PRISM computes the probability with which such a formula is satisfied, i.e. the value of $\mathrm{P}_{\sigma,\Phi} = \Pr\{\sigma \models \Phi\}$ for given σ and Φ. The models catered for by PRISM may incorporate specific probabilities for various behaviors and so may the

formulas used for verification. Probabilistic models and PRISM–like tools find applications in numerous areas of computer science where random behaviour is involved. Oft–cited applications are randomized algorithms, real–time systems and Monte Carlo simulation. The application of probabilistic model–checking to quantum systems is entirely appropriate, since quantum phenomena are inherently described by random processes; to reason about such phenomena one must account for this.

PRISM uses a built–in specification language based on Alur and Henzinger's REACTIVE MODULES formalism (see [23,37] for details). Using this language the user can describe probabilistic behaviour. Internally, a PRISM model is represented by a *probabilistic transition system.* In such a system, each step in a computation is represented by a *move,* or *transition,* from a particular state s to a distribution π of successor states. For technical details, refer to [37].

The probabilistic temporal logic PCTL [36] is used as the principal means for defining properties of systems modelled in PRISM. It suffices for our purposes to remind the reader of the meaning of the operator $\mathcal{U}$, known as "unbounded until". The formula $\Phi_1 \mathcal{U} \Phi_2$ expresses the fact that Φ_1 holds continuously from the current state onward, *until eventually* Φ_2 becomes **true**. The PRISM property $\mathrm{P} \geqslant 1[\Phi_1 \mathcal{U} \Phi_2]$ states that the formula $\Phi_1 \mathcal{U} \Phi_2$ is true with certainty, i.e. with a probability of unity; we use PRISM to check whether such a property holds in a given model.

6 Summary and Conclusion

This report has detailed various protocols of significance in practical QKD systems, in particular, reconciliation protocols (used for correcting errors after a QKD protocol has been performed), privacy amplification protocols (which are used for minimizing and if possible, eliminating, information gained by an attacker through eavesdropping on the public and private channel), and authentication protocols (which are used to verify the originator of a message on the two channels). All protocols described here are classical, in that they do not involve any quantum mechanical phenomena, as does QKD.

The second part of the report gives an account of the model–checking technique, which we believe to be adequate for modelling and analysing protocols such as the above. We discuss system specification (including the use of process calculus), temporal logic for property specification, and the features of the PRISM model checker. We expect that PRISM should be a powerful tool for investigating the behaviour of these protocols, bearing in mind that our research thus far has mainly involved the application of PRISM to a selection of purely quantum protocols [35,34].

References

[1] C. Boyd and A. Mathuria. *Protocols for Authentication and Key Establishment,* Springer–Verlag, 2003.

[2] L. Paulson. *The Inductive Approach to Verifying Cryptographic Protocols.* In *Journal of Computer Security* **6**, pp. 85–128, 1998.

[3] G. Lowe. *Breaking and Fixing the Needham–Schroeder Public Key Protocol using FDR.* In *Tools and Algorithms for the Construction and Analysis of Systems,* pp. 147–166. Springer–Verlag, 1996.

[4] Jörgen Cederlöf. *Authentication in quantum key growing.* Master's thesis, Dept. of Applied Mathematics, Linköpings Universitet, 1995.

[5] P. Ryan and S. Schneider. *Modelling and Analysis of Security Protocols,* Addison–Wesley, 2001.

[6] J. Mitchell, M. Mitchell and U. Stern. *Automated Analysis of Cryptographic Protocols using Murϕ.* In *Proceedings of the 1997 IEEE Symposium on Security and Privacy,* pp. 141–151. IEEE Computer Society Press, 1997.

[7] E.M. Clarke, S. Jha and W. Marrero. *Verifying Security Protocols with Brutus.* In *ACM Transactions on Software Engineering and Methodology* **9**(4), pp. 443–487, 2000.

[8] C. Meadows. *The NRL Protocol Analyzer: An Overview.* In *Journal of Logic Programming* **26**(2), pp. 113–131, 1996.

[9] M. Burrows, M. Abadi and R. Needham. *A Logic of Authentication.* In *ACM Transactions on Computer Systems* **8**(1), pp. 18–36, 1990.

[10] F.J.T. Fabrega, J. Herzog, and J. Guttman. *Strand Spaces: Why is a security protocol correct?* In *Proceedings of the 1998 IEEE Symposium on Security and Privacy,* pp. 160–171. IEEE Computer Society Press, 1998.

[11] D. Mayers. *Unconditional Security in Quantum Cryptography.* In *Journal of the ACM* **48**(3), pp. 351–406, 2001.

[12] C. Bennett and G. Brassard. *Quantum cryptography: Public Key Distribution and Coin Tossing.* In *Proceedings of the IEEE International Conference on Computers, Systems and Signal Processing, Bangalore, India,* pp. 175–179, 1984.

[13] M.N. Wegman and J.L. Carter. *New Hash Functions and their Use in Authentication and Set Equality.* In *Journal of Computer and System Sciences* **22**, pp. 265–279, 1981.

[14] C. Bennett. *Quantum Cryptography Using Any Two Nonorthogonal States.* In *Physical Review Letters* **68**(21), pp. 3121–3124, 1992.

[15] J.–M. Robert. *Détection et Correction d' Erreurs en Cryptographie.* Master's thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, 1985.

[16] C. Bennett, G. Brassard, and J.-M. Robert. *Privacy Amplification by Public Discussion.* In *SIAM Journal on Computing* **17**(2), pp. 210–229, 1988.

[17] C. Bennett, G. Brassard, and J.-M. Robert. *How to Reduce your Enemy's Information (Extended Abstract),* In *Proceedings of CRYPTO '85,* Lecture Notes in Computer Science **218**, pp. 468–476, Springer-Verlag, 1986.

[18] G. Brassard and L. Salvail. *Secret-key reconciliation by public discussion.* In *Advances in Cryptology — EUROCRYPT '93,* Lecture Notes in Computer Science **765**, pp. 410–423, Springer-Verlag, 1994.

[19] A. Ekert. *Quantum Cryptography based on Bell's Theorem.* In *Physical Review Letters* **67**(6), pp. 661–663, 1991.

[20] L. Salvail. *Le Problème de Réconciliation en Cryptographie.* Master's thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, 1991.

[21] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 1st edition, 2000.

[22] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Pearson Education, 2003.

[23] M. Kwiatkowska, G. Norman, and D. Parker. Modelling and verification of probabilistic systems. In P. Panangaden and F. Van Breugel, editors, *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems*. American Mathematical Society, 2004. Volume 23 of CRM Monograph Series.

[24] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985.

[25] R. Milner. *Communicating and Mobile Systems: The π–Calculus*. Cambridge University Press, 1999.

[26] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi–calculus. *Information and Computation*, 148:1–70, 1999.

[27] R. Nagarajan and S. Gay. Formal verification of quantum protocols. Available at arXiv.org. Record: quant-ph/0203086, 2002.

[28] S. Gay and R. Nagarajan. Communicating quantum processes. In *POPL '05: Proceedings of the 32nd ACM Symposium on Principles of Programming Languages, Long Beach, California*, January 2005.

[29] G. Holzmann. *The Design and Validation of Computer Protocols*. Prentice–Hall, 1991.

[30] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th* IEEE *Symposium on Foundations of Computer Science*. IEEE Press, 1977.

[31] M. Vardi. Branching vs. linear time: Final showdown. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 1–22. Springer–Verlag, 2001.

[32] N. Papanikolaou. Techniques for design and validation of quantum protocols. Master's thesis, Department of Computer Science, University of Warwick, 2005.

[33] N. Papanikolaou. Introduction to quantum cryptography. *ACM Crossroads Magazine*, 11.3, Spring 2005 Issue.

[34] R. Nagarajan, N. Papanikolaou, G. Bowen and S. Gay. An automated analysis of the security of quantum key distribution. CoRR Preprint cs.CR/0502048, available at `www.arxiv.org`.

[35] S. Gay, R. Nagarajan, and N. Papanikolaou. Probabilistic model–checking of quantum protocols. Quantum Physics Repository Preprint quant-ph/0504007, available at `www.arxiv.org`.

[36] F. Ciesinski and M. Größer. On probabilistic computation tree logic. In *Validation of Stochastic Systems* (2004), pp. 147–188.

[37] D. Parker, G. Norman and M. Kwiatkowska. PRISM 2.0 users' guide, February 2004.