Modeling and verification of the post-quantum key encapsulation mechanism KYBER using Maude

Víctor García[1,*], Santiago Escobar[1] and Kazuhiro Ogata[2]

[1]Universitat Politècnica de València (UPV), Camí de Vera, s/n, 46022 València, Valencia, Spain
[2]Japan Advanced Institute of Science and Technology (JAIST), Ishikawa 923–1292, Japan

Abstract

Communication and information technologies shape the world's systems of today, and those systems shape our society. The security of those systems relies on mathematical problems hard to solve for classical computers, that is, the available current computers. Recent advances in quantum computing threaten the security of our systems and the communications we use. In order to face this threat, multiple solutions and protocols have been proposed. Kyber is one of these protocols, and precisely it is a key encapsulation mechanism that bases its security in the learning with errors problem over module lattices. The presented work focuses on the analysis of Kyber to check its security under Dolev-Yao adversary assumptions. For that matter, we first learn about the current state of the solutions proposed against the threat of quantum adversaries and study how Kyber works. In the system-specification language Maude, we then construct a symbolic model to represent the behaviour of Kyber in a network. In this model, we conduct reachability analysis with the `search` command and find that a Man-In-The-Middle attack is present. Then we use the Maude LTL logical model checker to extend the analysis of the system by proving if liveness and security properties hold.

Keywords

Maude, rewriting logic, formal verification, post-quantum protocols, key encapsulation mechanisms

1. Introduction

Today's security is heavily based on complex problems. Most of the current network infrastructure and systems work over classical computers. Specifically, most of these protocols rely on three problems considered hard to solve under classic computation: the integer factorization problem, the discrete logarithm problem and the elliptic-curve discrete logarithm problem. Such problems are considered to be in category NP, which stands for non-polynomial time, for classic computers.

Research in the quantum field has been active in the past years, proposing new algorithms and methods that could endanger the security of current crypto-systems and cryptography protocols. As stated before, the protocols of today are based on mathematical problems hard to solve for classical computers, but such problems become solvable with quantum computers. Some of the

FAVPQC 2022: International Workshop on Formal Analysis and Verification of Post-Quantum Cryptographic Protocols, October 24, 2022, Madrid, Spain

*Corresponding author.
vicgarv2@upv.es (V. García); sescobar@upv.es (S. Escobar); ogata@jaist.ac.jp (K. Ogata)
http://personales.upv.es/sanesro/ (S. Escobar); http://www.jaist.ac.jp/~ogata/ (K. Ogata)
0000-0003-0681-1130 (V. García); 0000-0002-3550-4781 (S. Escobar); 0000-0002-4441-3259 (K. Ogata)

most popular asymmetric (or public key) algorithms, which rely on integer factorization, will become insecure under quantum computers using Shor's algorithm [1]. Another example is Grover's search algorithm [2] for unstructured databases, which in principle one could ask what it has to do with cryptography. It has been shown in that same paper that this algorithm makes it possible to reduce the complexity of the integer factorization problem to a quadratic cost.

In order to face the threat that quantum computers suppose to the security of most information systems, the National Institute for Standards and Technologies (NIST) started in 2017 the Post-Quantum Cryptography Project (PQC). The project is conducted as a competition, divided into multiple rounds, to analyze candidate protocols to be used in a standard as a solution against quantum adversaries. There have been four rounds of the project, and the candidates vary from public-key encryption and key-establishment algorithms to digital signature algorithms. For this work we selected round 3, which was finalized in 2020. Round 4, announced on the 5th of July of 2022, marks the near end of the project. This paper focuses on public-key encryption and key-establishment algorithms because some attacks have been found, such as Man-In-The-Middle or Meet-In-The-Middle (MITM) attacks.

The protocol we selected to work with is Kyber, more specifically CRYSTALS-Kyber, from the suite of protocols denoted as CRYSTALS, standing for Cryptographic Suite for Algebraic Lattices. Kyber is a Key Encapsulation Mechanism (KEM) and bases its security on the hardness of solving the Learning With Errors problem over module lattices. These KEMs' primary goal is to securely share a given key between two network participants where channels are not safe from intruders. Such a goal is interesting for conventional cryptography, also known as Symmetric Cryptosystem, which uses a secret key to encrypt a message. We deeply explain Kyber in Section 3.

Now that we have the problem of quantum computers and a possible solution provided by NIST, we need to establish how to analyze and reason about the protocol. For the analysis of security systems and protocols, two kinds of approaches can be taken: computational security and symbolic security. The former is based on mathematical proofs over a computational model, where messages are bit strings, and the adversary is any probabilistic Turing machine. Cryptographers generally use computational security, and the authors of Kyber have already covered this approach. The latter is based on the use of symbols, where the cryptography primitives are function symbols acting as black boxes. It is important to note that these models assume perfect cryptography, i.e., ciphertexts cannot be broken without the proper key. Although the computational model is closer to reality, it complicates the proofs and is hard to understand for non-experts of cryptography. On the other hand, symbolic models are suitable for automation and easier to understand, so in this paper our experiments belong to the latter. It is essential to mention that this approach not only can be applied to the selected protocol but also any other scheme or mechanism on rounds 3 or 4 of the Post-Quantum Cryptography project.

Related work: Current advances in the security of protocol analysis have been made. One interesting idea is the one proposed at [3], where the author explains several examples about the formal specification of protocols and introduces and explains the symbolic and computational model analysis. In [4], the authors explore the current literature and papers on both symbolic and computational analysis of protocols. In this survey, they analyze the results by combining both types of analysis. This proposal was made initially by [5] in order to close the gap

between both lines of protocol verification. One can also find specific papers [6][7] applying symbolic, computational or both analyses over cryptography protocols. In the former, the authors apply symbolic analysis, specifically Automated Theorem Proving (ATP), to verify the IKEv2 handshake protocol from the suite IPsec, finding security gaps. In the latter, the authors present a variant of a handshake protocol from the WireGuard VPN protocol with post-quantum capabilities. They perform such adaptation by replacing the previous Diffie-Hellman-based handshake with key-encapsulation mechanisms (KEMs). The authors verify their proposal's security with symbolic and computational proofs. On the one hand, the symbolic proofs can verify more security properties than the computational proofs and are computer verified. On the other hand, the computational proofs give stronger security guarantees as the proof makes less idealizing assumptions.

Among the various protocol analysis tools available, we have Maude-NPA [8], related to the programming language Maude [9]. Maude-NPA has a theoretical basis on rewriting logic, unification and narrowing and performs a backwards search from a final attack state to determine whether or not it is reachable from an initial state. Other tools, such as ProVerif [10], are based on an abstract representation of a protocol using Horn clauses. The verification of security properties is done by reasoning on these representative clauses. Other tools such as Tamarin [11] are based on constraint solving to perform an exhaustive, symbolic search for executions traces. Furthermore, other tools such as Scyther [12] or CPSA [13] attempt to enumerate all the essential parts of the different possible executions of a protocol. Also the DEEPSEC prover [14] is another tool which is mostly used to decide equivalence properties.

Finally, [15][16] are first symbolic security analyses of a collection of post-quantum protocols, among which Kyber is found. A man-in-the-middle (MITM) attack is found for each of those protocols, with the search analysis using Maude. This paper aims to provide a higher level or more abstract model and symbolic analysis of Kyber than [15]. A more abstract model is needed to better understand and reason on KEMs for non-experts of cryptography. Our symbolic analysis also aims to demonstrate that a MITM attack is present and prove that our model considers all protocol's possible behaviours. In [16], the same authors have used the methodology from [15] for the KEM known as SABER, a close relative of Kyber. It may be interesting to apply our new methodology proposed in this paper to other KEMs like SABER.

Roadmap: The rest of the paper is structured as follows. Sect. 2 introduces the Dolev-Yao adversary model and a gentle presentation of what Maude is. Sect. 3 explains the behaviour and security principles of Kyber. Sect. 4 describes the core modules for the construction of our symbolic model of the protocol. Sect. 5 dives into the two approaches carried on over the symbolic model. Finally, Sect. 6 summarizes the paper.

2. Preliminaries

2.1. Dolev-Yao adversary model

The Dolev-Yao adversary model was first introduced in [17]. In this paper, the authors explained that public-key schemes are secure against adversaries that cannot modify the environment, which is unrealistic. That is why they presented different examples of protocols whose security properties could be compromised if an intruder can take action over the messages of a network.

An intruder can be either passive or active over a network where other participants send and receive messages during, for example, a handshake protocol or a key exchange scheme. The former intruder can only read the message and extract raw content from it, meaning they cannot derive any information from messages without the proper private key. The latter type of intruder cannot only read messages but also modify them and send them through the network. It is essential to clarify that the intruder is considered a polynomial-time Turing machine.

In this seminal work, the authors proposed the Dolev-Yao intruder model. This model states the capabilities an intruder has over a network. Such capabilities are:

- Intruder can obtain any message that is passing through the network.
- He is a *legitimate* user of the network. That is, he can do any actions a honest participant can.
- The intruder has the opportunity to be a receiver to any participant. That is, he can receive messages from other participants.

It must be noted that the network participants, the intruder included, must comply with the following assumptions.

- One-way functions are unbreakable. In other words, the basic primitives of the protocol are considered to be non-reversible.
- The definition of the protocol cannot be changed and must be followed by any participant. A user cannot do undefined steps during the protocol execution.
- Public keys can be used for encryption by everyone.
- Private keys can be used for decryption of messages encrypted by the corresponding public key.

2.2. Maude

Maude [9] is based on rewriting logic [18], a logic ideally suited to specify and execute computational systems in a simple and natural way. Since nowadays most computational systems are concurrent, rewriting logic is particularly well suited to specify concurrent systems without making any a priori commitments about the model of concurrency in question, which can be synchronous or asynchronous, and can vary widely in its shape and nature: from a Petri net [19] to a process calculus [20], from an object-based system [21] to asynchronous hardware [22], from a mobile ad hoc network protocol [23] to a cloud-based storage system [24], from a web browser [25] to a programming language with threads [26], or from a distributed control system [27] to a model of mammalian cell pathways [28, 29]. And all *without any encoding*: You see and get a direct system definition without any artificial encoding.

Maude is based on rewriting logic, and rewriting logic has a sub-logic called membership equational logic. This sub-logic defines a system's deterministic parts using functional modules. In contrast, Maude system modules represent concurrent systems as conditional rewrite theories that model a nondeterministic system which may never terminate and where the notion of a computed value may be meaningless. In this concurrent system, the membership equational sub-theory defines the states of such a system as the elements of an algebraic data type, such as

terms in an equivalence class associated with cryptography properties. We can call this aspect the static part of the specification. Instead, its dynamics, i.e., how states evolve, are described by the transition rules, which specify the possible local concurrent transitions of the system thus specified. The system's concurrency is naturally modeled by the fact that several transition rules in a given state may be applied concurrently to different sub-parts, producing several concurrent local state changes. Thus rewriting logic models those concurrent transitions as logical deductions [18].

The most basic form of system analysis, in the form of explicit-state model checking, is illustrated by the use of the `search` command in Maude that performs reachability analysis from an initial state to a target state. Reachability can be used to verify invariants or find violations of invariants in the following sense. We can search for a violation of an invariant. If the invariant fails to hold, it will do so for some finite sequence of transitions from the initial state, which will be uncovered by the `search` command above since all reachable states are explored in a breadth-first manner. If the invariant does hold, we may be lucky and have a finite state system, in which case the `search` command will report failure to find a violation of the invariant. However, if there is an infinite number of states reachable from the initial state, the search will never terminate.

Under the assumption that the set of states reachable from an initial state is finite, Maude also supports explicit-state model checking verification of any properties in linear-time temporal logic (LTL) through its LTL model checker.

3. Key Encapsulation Mechanism KYBER

3.1. Behaviour

In order to explain the behaviour of Kyber, we should look at Fig. 1. Participants of the protocol will be Alice and Bob for literature reasons.

The protocol is initiated when a participant, id est, Alice, performs the *KeyGen* step. *KeyGen* is a function which has no parameters and will provide a pair of keys *(PK, SK)*. The former key, *PK*, is the public key and can be known by every other network participant. The latter, *SK*, is a secret key only known and accessible by the user that generated it, in this case, Alice. Once Alice has both keys in his possession, she will send a message to another participant, Bob, with her public key. Once Bob receives Alice's public key, he performs *Enc*, which stands for *Encrypt*, using the received public key. Function *Enc* produces a pair *(C, K)*, where *C* is a ciphered text containing the second element of the pair, *K*, which is the future shared key between the two participants. Once Bob has in his storage the key and the ciphered text, he sends the latter in a new message back to Alice, who started the protocol and from whom he used the public key. Alice then receives the ciphered text *C* from Bob and uses his secret key *SK* to perform *Dec* function over *C*. *Dec* outputs ideally the original shared key generated by Bob that is contained in *C*. In the last step, both participants securely shared a key *K* between them.

As we have seen, the network is elementary, and participant interaction is minimal. No confirmation messages or any prior establishment to know where the participants are in the network is performed. Such discovery and set-up work are assumed to have been done previously.

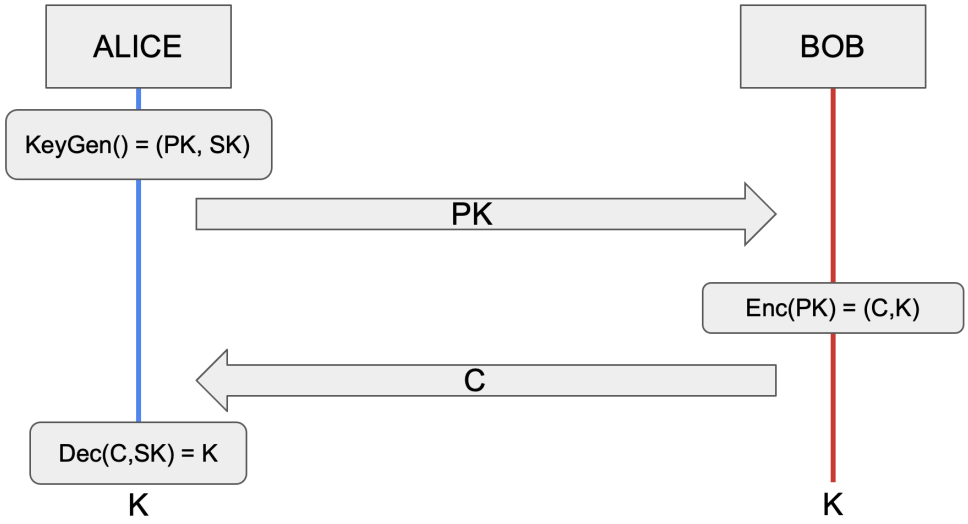

Figure 1: High level view of the behaviour of Kyber between two honest participants, id est, Alice and Bob.

Finally, about the used cryptography primitives of *KeyGen*, *Enc* and *Dec* we must remark that there are two versions for each of them, as we will see in the next section. Recall that these functions are treated as black boxes.

3.2. Security fundamentals

Let us now see why these primitives, which we have seen at a high level for key generation, encryption and decryption, are resistant to the computational capacity of a quantum adversary. Kyber is an IND-CCA2-secure key encapsulation mechanism (KEM) whose security is based on the hardness of solving the learning-with-errors (LWE) problem over module lattices. Kyber works with vectors and matrices of polynomials with various operations, such as concatenation, transposition, product or other more complex ones such as hash and key derivation functions. These operations can be seen in Fig. 3 and are present in the three main functions.

Here in Fig. 2, it should be noted that in *KYBER.Dec(c,sk)* when the sub-function CPAPKE.Dec(c,$\mathbf{s}$) occurs, the computed text m' could not be the same as the one generated by the other participant in KEM.Enc(pk) with sub-function CPAPKE.Enc(pk,m,r). This different message m' is a value close to m given the property *isSmall(p)* over a polynomial p. We say that a polynomial p is *small* when its degree is lesser than a given number established by the protocol. The close value m' is then used to compute a new value, but also close, c' which is compared to the received c in a message. Depending on their equality, the construction of the shared key will be different with the key derivation function (KDF). This differentiation of values arise with low probability, but it states that encryption and decryption phases are not error-prone.

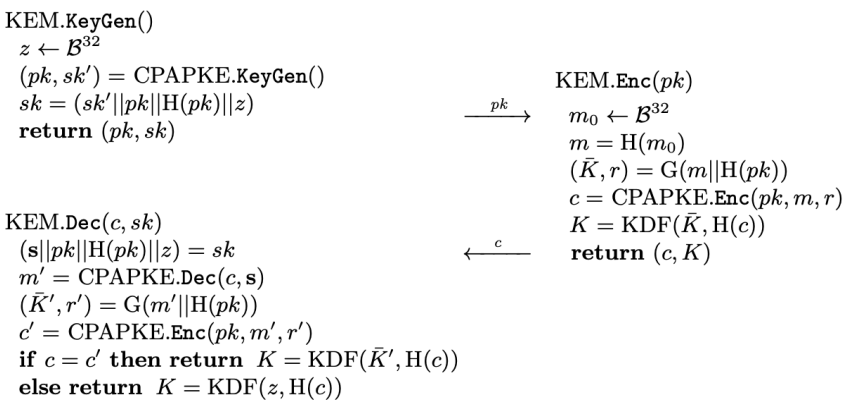

Figure 2: Algorithms of Kyber Key Encapsulation Mechanism borrowed from [15].

Nevertheless, why are they different in the end? Well, here lies the strength of the scheme against a quantum adversary. If we check the most internal functions, that is, the CPAPKE ones shown in Fig. 3, we can see in algorithm CPAPKE.Enc that there are vector values such as $\mathbf{e}_1$ and e_2 that have been sampled with a random seed r using function sampleCBD from a centered binomial distribution. Such errors elevate the computational complexity of the scheme from polynomial time (P) to non-polynomial time (NP), making it unsolvable by quantum computers because of the randomness.

$$X' = Decompress_q(Compress_q(X, 1), 1) \tag{1}$$

The error is tried to be eliminated by function Decompress$_q$ when extracting from the pair of vectors c_1 and c_2 the vectors $\mathbf{u}$ and v respectively, at the middle of step CPAPKE.Dec. This function has a property in combination with function Compress$_q$, which states in Eq. 1 that decompressing the compress of a given value X with the same second parameters, gives a new value X' which is similar to the original compressed value. This property takes place while computing Compress(v'-$\mathbf{s}^T\mathbf{u}$',1).

It is important to mention that other operations take place such as *generate*, *sampleCBD*, *Compress*, *Decompress*, *encode* and *decode*. These are necessary for the main three operations we have described before but are not explained in detail in this paper because they are not necessary for our understanding of the protocol. Full descriptions of these specific functions are available at [30].

4. Maude specification

4.1. Assumptions

On the specification of the symbolic module for Kyber, we have taken the freedom to make some assumptions about it. We can divide these assumptions into Dolev-Yao adversary assumptions and Kyber-specification assumptions.

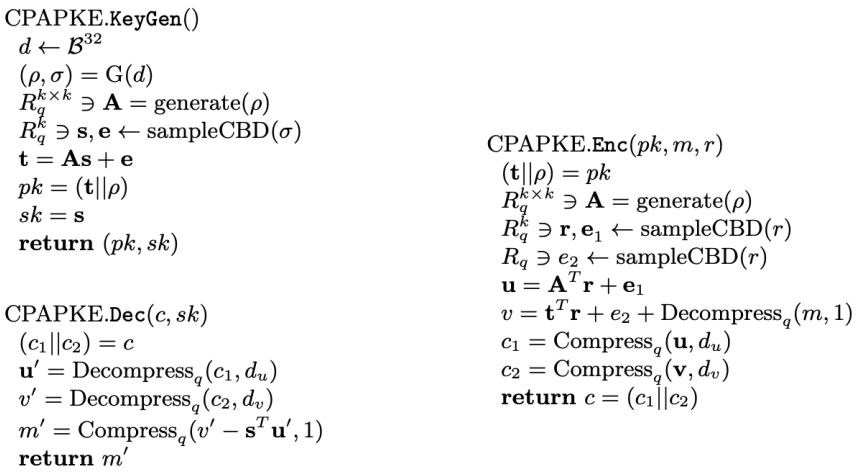

Figure 3: Internal algorithms of the Kyber Key Encapsulation Mechanism borrowed from [15].

We will start with the former one because we will be importing all of the characteristics the Dolev-Yao adversary model states. This implies additional rules and conditions for our system module. These new rules are explained in the following Section 4.2. We also assume that there will be only three participants in the network, two of them honest (Alice and Bob) and one adversary (Eve).

About Kyber, almost all assumptions are over mathematical and low-level concepts. Such assumptions allow us to abstract our model from implementation requirements and focus on representing the desired behaviour among the participants.

- All matrices are square matrices.
- All vectors are column vectors, and transposed vectors are row vectors.
- All vectors, which also represent polynomials, are considered to be of the necessary degree to be considered small, thus fulfilling the property *isSmall(p)* explained in Section 3.2.
- We only consider for decompression function the ideal case where there is no error in obtaining *m*, thus *m'* will be equal to *m*.
- We assume that the deciphered message is the shared key between the participants, so no additional functions, KDF in this case, need to be specified and then applied.
- Our sampling procedures are deterministic, but we will assume that the operators sampled are from a CBD whenever it is the case.

4.2. Module composition

The symbolic model of Kyber is composed of various modules, precisely five functional modules and one system module. Functional modules are `DATA-TYPES`, `KYBER-HASH-OPERATIONS`, `ENCRYPTION`, `KYBER-CPAPKE-KEYGEN` and `KYBER-CPAPKE-ENC`. The system module representing the protocol is `KYBERV2`.

4.2.1. Functional modules

DATA-TYPES: About this functional module, we should remark on the importance of all mathematical assumptions. Specifically, the three first assumptions found in the previous section apply to this module. The module represents the low-level components of our symbolic model toward the official specification. Here, we represent matrices and vectors and all their associated operations. Vector operations as the concatenation, addition and subtraction of vectors are declared and defined in this module. First, the concatenation of two vectors is a new vector defined with the symbol `||`. On the one hand, vector sum is declared with symbol `v+` being commutative and associative with the identity element `0`. On the other hand, the subtraction of two vectors is defined with symbol `v-` as associative. Finally, we declared the *dot* product as associative only.

Changing to matrices, we declared and defined the product operation between a matrix and a vector, resulting in a new vector. The product of matrices and vectors is associative, and we defined the following property over the *dot* product with the Eq. 2. Symbols *V1* and *V2* are vectors, and *M2* is a matrix.

$$(M1\ m*\ V1)\ dot\ V2\ =\ V1\ dot\ (M1\ m*\ V2) \tag{2}$$

The transpose function is defined for vectors and matrices. Their distributive properties over operations such as product or addition of vectors and matrices are formalized in the Eq. 3. Once again, the symbols are the same as in the previous equation, but now we specify two distribution cases. The upper equation defines the distribution of the transpose of a vector over the sum of vectors. The lower defines the distribution of the transpose of a vector over the matrix-vector product.

$$\begin{aligned} tpV(V1\ v+\ V2)\ &=\ tpV(V1)\ v+\ tpV(V2) \\ tpV(M1\ m*\ V1)\ &=\ tpV(V1)\ m*\ tpM(M1) \end{aligned} \tag{3}$$

We also define in functional module *DATA-TYPES* the concept of *Pair* of vectors. We declare and define functions *first* and *second* which receive a pair and return the first and second elements respectively.

It is important to mention that, in this module, we define some constants for values used in the protocol. Constants *du* and *dv* are used in the compression and decompression of u and v respectively. These two constant values are non-zero natural numbers that take a specific value depending on the version of Kyber they are used. For us, they are constants to represent such value in a symbolic form. We also define three pairs of constants for *Rho* and *Sigma* in order to be able to sample different values in future modules.

ENCRYPTION: This functional module is one of the most important of our symbolic model because it specifies our equational theory to represent the compression and decompression properties shown in Subsection 3.2. Equations available in Eq. 4 represent the property of decompressing compressed content and vice versa. The commutativity of the given property is necessary because the protocol applies Compress over Decompress, even if the property is written otherwise in the mathematical explanation. Within these equations, *X*, *V1* and *V2* are vectors, and *N* is a natural number different from zero.

```
eq  (V1 v+ Decompress(X,N)) v- V2 = Decompress(X,N) .
```

Figure 4: Definition of the property to cancel the noise present in the component of a ciphered text c in functional module *ENCRYPTION*.

```
ops Alice Eve Bob : ->  Identifier .
op _[_]_ : Identifier Keys Content -> Principal .
```

Figure 5: Definition of a participant at the network in our system module KYBERV2.

$$\begin{aligned} Decompress(Compress(X, N), N) = X \\ Compress(Decompress(X, N), N) = X \end{aligned} \tag{4}$$

We also specify the property of noise cancellation that represents the ideal case where both noises from u' and v' in the Eq. 5 are properly cancelled because of subtraction. The following Fig. 4 shows how we modelled it so that when we have a compressed value and two vectors, they cancel each other as the noise equation specified.

$$Compress(v' - s^T u', 1) \tag{5}$$

4.2.2. System module

With our system module identified as *KYBERV2*, we try to model the behaviour of honest participants and the capabilities the intruder has over the network. First, we will define representations of elements in our model for participants, messages and the global state of the system. Then participant behaviour is explained and finally we move to the definition of intruder capabilities.

Element definitions: Participants in our model follow the operational definition shown in Fig. 5. Here three identifiers are declared for our corresponding participants. Then the structure of a participant is declared. A participant consists of an *Identifier*, like the ones that have been defined, a group of keys the participant know, and a group of elements that represent its elements that are not keys and the participant has in its possession, that is, in its memory.

Messages are defined by the operator *msg* shown in the code below. A message contains information about two participant identifiers, the status of the message and the content it carries. The first identifier indicates the source of the message, and the second is the identifier of the participant to whom the message is delivered throw the network. Then, the status of a message can take the values *sentX*, *receivedX* and *interceptedX*, where X can either be *PK* or *C* depending on the step. At last, the content of the message is assumed to be secure, meaning a participant can not infer any additional contents from it without the required information.

```
op msg{(_,_)[_]_} : Identifier Identifier MsgState Content -> Msg .
```

```
crl [KeyGen] : { ds(SAM1 CONT emptyS) CONT2 } < (ID1[emptyK]emptyC) PS >net(MSGS) =>
{ ds(CONT emptyS) CONT2 } < (ID1[publicKey(ID1,PK) ; secretKey(ID1,SK)]dI(ID1,SAM1)
emptyC) PS >net(MSGS)
    if SK := sampleS(second(G(SAM1))) /\
    PK := ((generateA(first(G(SAM1))) m* SK) v+ sampleE(second(G(SAM1)))) .
```

Figure 6: Definition of conditional rule *KeyGen* in our system module KYBERV2.

The definition of the structure representing our system can be seen in the code below. Here, and from left to right, are specified the elements our rules will handle. At the right corner, we assigned a field for all the available sample values, that is, constant values representing vector values, to use them in *Keygen* and *Enc* functions. Then we have a pool of participants, following each one the structure previously explained. Then, at the left end, we have the network, with a pool for associative messages, representing a kind of record that lets participants work over the sent messages.

```
op {_}<_>net(_) : Content Principals Msgs -> GlobalState .
```

Participant behaviour: Following the Kyber specification [30] and the Dolev-Yao assumptions, we specify the following rules in Maude to model how the protocol operates. All these rules have been written to be as general as possible, making the model and the constructed execution tree more realistic and compelling for model checking.

The first rule is *KeyGen* as can be seen in Fig. 6. This rule is the one that starts the protocol for a given participant with identifier `ID1`. The rule states that given a participant with identifier `ID1` whose content is empty both for the keys and memory, can generate a `publicKey(ID1,PK)` and a `secretKey(ID1,SK)` in the group of keys. This is possible if there is a value `SAM1` in the sample group for *d*, which is stored in the content of the participant using the operator *dI* that associate a sample value with a participant identifier. The construction of both keys, public and secret, is done through the matching equations in the rule's conditions. The structure is the one present at the specification and can be seen in Fig. 3, where public key *PK* is the matrix *A* multiplied by the secret key *s* plus a sampled error *e*. For the secret key *s*, we assumed it to be just the sample value from the CBD, so no further operations are needed for its computation.

We also defined a rule *SendPK* that models the behaviour of a participant with his public key, sending it to any other participant in the network different from him. The message is sent if it has not been sent previously, so we avoid infinite execution.

Then, to complement the previous rule, we defined rule *ReceivePK* to process the last incoming message if it contains a public key and has not been received yet. The reason to only check the last message is to have some kind of history or log over the messages in the network.

Rule *Enc* as it is shown in Fig. 7 models the function with its same name. In order to apply the encryption step, a participant first has to receive the public key from the other peer. The participant also needs to be able to sample values for the message *m* to transmit and a random *'coin' r* which is a value used in sampling the errors e_1 and e_2. In application of the rule, the participant possesses in the pool of keys a new shared key containing the value to be transmitted

```
crl [Enc] : { ms(SAM1 CONT1 emptyS) rs(SAM2 CONT2 emptyS) CONT3 } < (ID2[
publicKey(ID1,(M1 m* V1) v+ V2) ; KS2] CONT4) PS >net(MSGS) => { ms(CONT1 emptyS)
rs(CONT2 emptyS) CONT3 } < (ID2[sharedKey(ID1,SAM1) ; publicKey(ID1,(M1 m* V1) v+ V2)
; KS2]cI(ID2,C) mI(ID2,SAM1) rI(ID2,SAM2) CONT4) PS >net(MSGS)
        if U := ((tpM(M1) m* sampleR'(SAM2)) v+ sampleE1(SAM2)) /\
        V := (((tpV((M1 m* V1) v+ V2) dot sampleR'(SAM2)) v+ sampleE2(SAM2))
        v+ Decompress(SAM1,1)) /\
        C1 := Compress(U,du) /\ C2 := Compress(V,dv) /\ C := (C1,C2) /\ ID1 =/= ID2 .
```

Figure 7: Definition of conditional rule *Enc* in our system module KYBERV2.

```
crl [Dec] : { CONT } < (ID1[secretKey(ID1,SK) ; KS1]dI(ID1,SAM1) cI(ID2,C') CONT1)
(ID3[KS2]cI(ID3,C') mI(ID3,SAM2) rI(ID3,SAM3) CONT2) PS >net(MSGS) => { CONT } <
(ID1[secretKey(ID1,SK) ; sharedKey(ID2,K1) ; KS1]dI(ID1,SAM1) CONT1)
(ID3[KS2]cI(ID3,C') mI(ID3,SAM2) rI(ID3,SAM3) CONT2) PS >net(MSGS)
        if SK := sampleS(second(G(SAM1))) /\
        PK := ((generateA(first(G(SAM1))) m* SK) v+ sampleE(second(G(SAM1)))) /\
        U := ((tpM(generateA(first(G(SAM1)))) m* sampleR'(SAM3)) v+ sampleE1(SAM3)) /\
        V := (((tpV(PK) dot sampleR'(SAM3)) v+ sampleE2(SAM3)) v+ Decompress(SAM2,1)) /\
        C1 := Compress(U,du) /\ C2 := Compress(V,dv) /\ C := (C1,C2) /\
        U' := Decompress(first(C),du) /\ V' := Decompress(second(C),dv) /\
        K1 := Compress(V' v- tpV(SK) dot U',1) /\ ID1 =/= ID2 .
```

Figure 8: Definition of conditional rule *Dec* in our system module KYBERV2.

securely to the other participant. In the content pool, the ciphered text containing such a shared value is stored for later. As in *KeyGen* rule, the conditions of it are used to construct the needed cryptography elements, in this case a ciphered text c consisting of a pair of ciphered texts c_1 and c_2. Both elements are specified following the operations in Fig. 3.

The counterpart of *SendPK* but for a ciphered text c obtained in *Enc* is the conditional rule *SendCiph*. It checks similar conditions when sending the public key, so no infinite execution happens.

Then, the rule to receive a ciphered text, behaves similarly to its counterpart, that is, *ReceivePk*. The rule applies when there is a sent message in the network for a given participant with *ID1*. The content of the message is stored by the participant in its pool of content, acting as memory. As with its counterpart *ReceivePK*, our rule only checks the last sent message on the network for modelling reasons.

Finally, the rule to decipher the received cryptogram is *Dec*. In Fig. 8 we can see that the computation `V' v- tpV(SK) dot U'` is performed as is specified in Section 3.2. This operation cancels the noise *U'* and *V'* have, leaving alone *Decompress(SAM1,1)* so it can be extracted *SAM1* when Compress is applied, obtaining the shared key *K1*.

Intruder behaviour: When specifying the intruder's capabilities over our module, we decided to specify two rules, *Intercept1* and *Intercept2*.

```
crl [Intercept1] : { CONT } < (Eve[publicKey(Eve,PK) ; KS1]CONT1) (Alice[publicKey(Alice,PK')
; KS2]CONT2) PS >net(MSGS msg{(Alice,Bob)[sentPK]PK'}) => { CONT } < (Eve[publicKey(Eve,PK) ;
publicKey(Alice,PK') ; KS1]CONT1) (Alice[publicKey(Alice,PK') ; KS2]CONT2) PS >
net(MSGS msg{(Alice,Bob)[interceptedPK]PK'} msg{(Alice,Bob)[sentPK]PK})
        if (msg{(Alice,Bob)[interceptedPK]PK'}) in MSGS == false .
```

Figure 9: Definition of conditional rule *Intercept1* in our system module KYBERV2.

```
crl [Intercept2] : { CONT } < (Eve[publicKey(Eve,PK) ; KS1]cI(Eve,C') CONT1) PS >net(MSGS
msg{(Bob,Alice)[sentC]C}) => { CONT } < (Eve[publicKey(Eve,PK) ; KS1]cI(Eve,C')
cI(Bob,C) CONT1) PS >net(MSGS msg{(Bob,Alice)[interceptedC]C} msg{(Bob,Alice)[sentC]C'})
        if (msg{(Bob,Alice)[interceptedC]C}) in MSGS == false /\ C =/= C' .
```

Figure 10: Definition of conditional rule *Intercept2* in our system module KYBERV2.

The former can be seen in Fig. 9, and it binds the intruder with the ability to intercept a sent message containing a public key. The intercepted message is marked with a new status representing its deceased ability to be received. The intruder also places a new message identical to the previous one but with his public key as content. This change makes the receiver think the public key received is from the sender when it is not, thus beginning the man-in-the-middle attack.

The latter is available in Fig. 10 and makes the intruder intercept a message sent with a ciphered text. This intercepted message is sent by the receiver from the previous fake message and makes *Eve* send a new message but with his own ciphered text. In this way, *Eve* has in store two ciphered texts, his own and the one intercepted.

5. Maude verification

5.1. Reachability verification

We will verify, using the `search` command, if the model behaves as expected, which means checking if states of interest exist. For that goal we conduct reachability analysis from two initial states, `init1` and `init2`, both available in Fig. 11. The initial state `init1` defines our global state with three types of samples, each with a value available for sampling. It also specifies that three participants populate the network, and the network of messages is initially empty. The other initial state `init2` is the same as the first but with the difference that there is one more sample value for each of the possible samples of `ds`, `ms` and `rs`.

For each of these initial states, we checked two things:

- If there exists a state where two participants have successfully shared a key. It is achieved with the command: `search initX =>* True .`, where `initX` is one of the initial states and `True` is a constant value, obtained through application of rule *Comp* seen below. This constant of sort *global state* represents that two participants have succeeded in the application of the protocol, hence they have shared a key.

```
eq init1 = {ds(d1 emptyS) ms(m1 emptyS) rs(r1 emptyS)} < (Alice[emptyK]emptyC)
(Eve[emptyK]emptyC) (Bob[emptyK]emptyC) >net(emptyM) .

eq init2 = {ds(d1 d2 emptyS) ms(m1 m2 emptyS) rs(r1 r2 emptyS)} < (Alice[emptyK]emptyC)
(Eve[emptyK]emptyC) (Bob[emptyK]emptyC) >net(emptyM) .
```

Figure 11: Definition of two initial states for our system module KYBERV2 in Maude.

```
search initX =>* { CONT } < (ID1[sharedKey(ID3,K1) ; KS1]CONT1)  (ID2[sharedKey(ID1,K1) ;
    sharedKey(ID3,K2) ; KS2]CONT2) (ID3[sharedKey(ID1,K2) ; KS3]CONT3) >net(MSGS)  .
```

Figure 12: Command template for a man-in-the-middle attack search in Maude.

- If a state exists in the state space tree in which a man-in-the-middle attack has happened. It is achieved with the command in Fig. 12, where the final state specifies that there are three participants in the global state, and the shared key between `ID1` and `ID2` is the same, and a key has been shared between `ID2` and `ID3`. The trick here is that `ID1` and `ID3` think they have shared the same key, thus resulting in a man-in-the-middle attack.

```
rl [Comp] : { CONT } < (ID1[sharedKey(ID2,K1) ; KS1]CONT1)
    (ID2[sharedKey(ID1,K1) ; KS2]CONT2) PS >net(MSGS) => True .
```

On the first initial state, the result of applying the first `search` commands is that there is a state where two participants have shared a key, meaning the protocol works. Moreover, the second search does not find a solution, stating that no man-in-the-middle attack is found when there are only sufficient sample values for one key exchange of the protocol.

Regarding the second initial state the results of both search commands are quite interesting. As well as in the first state, state two lets two participants share a key securely between them, but with the second search, a man-in-the-middle attack is found. The second search returns multiple solutions given the model's different possibilities in message passing, but we show only the first solution. This solution states that two honest participants, *Alice* and *Bob* have shared keys for each of them that are different, and the values match with the ones the third participant, the intruder *Eve*, has in his possession. So when *Alice* or *Bob* try to use those secret keys to communicate with each other, they will indeed be sending messages to *Eve* thinking it is the other honest participant. We can also see in the message section that some of them have been intercepted by *Eve*.

5.2. Formal verification

5.2.1. Predicates

In order to use model checking in Maude, one needs two things: a system module representing the system and some predicates to define the properties of our system. The system module has already been defined and tested, and in this subsection, we dive into the three predicates we have specified.

Predicate *wantsToShareKey* is defined so it is true in a global state where a participant with identifier *ID1* has his own public key, meaning he has performed *KeyGen* step, and there is a message to another participant, with identifier *ID2*, different than him. This predicate represents a participant wanting to share a key with another. In other words, it is the start of the protocol Kyber.

```
eq { CONT } < (ID1[publicKey(ID1,PK) ; KS1]CONT1) (ID2[KS2]CONT2) PS >
    net(MSGS msg{(ID1,ID2)[sentPK]PK}) |= wantsToShareKey(ID1,ID2) = true .
```

Predicate *sharedAKeyWith* is defined so it is true when two participants hold the same shared key for the other one. This predicate represents the end of the protocol, fulfilling the previous predicate in which the protocol was started.

```
eq { CONT } < (ID1[sharedKey(ID2,K1) ; KS1]CONT1) (ID2[sharedKey(ID1,K1) ; KS2]CONT2)
    PS >net(MSGS) |= sharedAKeyWith(ID1,ID2) = true .
```

Finally, the last predicate *stolenSecret* is defined to state that a secret key has been stolen from a participant if it exists in the pool of keys of a different participant.

```
eq { CONT } < (ID1[secretKey(ID1,SK) ; KS1]CONT1) (ID2[secretKey(ID1,SK) ; KS2]CONT2)
    PS >net(MSGS) |= stolenSecret(ID1,ID2) = true .
```

5.2.2. Properties

With the predicates defined, we now specify two properties. One checks the fairness of our protocol, and the other has to do with its security. These properties are LTL formulas that allow us to explore the execution tree in search of counterexamples that do not satisfy the formulas, thus proving the property does not hold. If no counterexample is found, we can say with assurance that the symbolic model satisfies the given property.

The first property, *FAIRNESS*, has to do with the assurance that once a participant wants to share a key with another participant, both in the end agree on one. It can be noted that the property is written for only the case when the participants are *Alice* and *Bob*, that is, the honest parts of the network.

The second property, *SECURITY*, has to do with the assurance that the predicate of *stolenSecret* is true in any future state. In other words, no participant learns the secret key of another one. Note that the property is only specified for the case when the secret key is from *Alice* and the thief is *Eve*.

5.2.3. Results

About the execution of our LTL formulas, we have applied both over our two initial states: `init1` and `init2`. The results are that both initial states accomplish the fairness property when *Alice* and *Bob* want to share a key. We can also conclude that the security of the secret key only to be known by its owner is assured thanks to the model checker not finding any state where *secretStolen* holds when *Alice* is the key holder and *Eve* the thief.

6. Conclusion

In this paper, we have proven the presence of a man-in-the-middle attack on the key encapsulation mechanism Kyber given Dolev-Yao adversary assumptions. Moreover, two LTL formulas specifying both fairness and security properties have been applied with Maude's LTL Logical Model Checker to extend our model's verification. Our results on a symbolic model that represents Kyber prove that its last version is not safe from classical adversaries if no authentication is available or defined, thus breaking the scheme. Finally, with this new symbolic model, we have provided a new approximation to represent the system different from [15]. We also have provided and applied a new analysis approach using the model checker, thus extending the reachability analysis made by [15].

For future work, we plan to improve the model by conducting extended model checking to verify its complete correctness. We also want to extend the model to represent the key encapsulation mechanism better. In the future, there could even be the possibility to specify multiple layers of protocols and check the interaction between them. For example, add capabilities of authentication or signatures to Kyber and perform the analyses we have made, in this paper, to check if the results are the same. We could also extend the system representation to use the objects feature from Maude, so it is closer to other languages and even more understandable for non-experts in formal methods. We are also considering using protocol analysis tools, such as Maude-NPA, to specify the protocol and check its security in a more thoughtful analysis form of the system. These new tools will also let us check if an attack is present for an unbounded number of sessions.

References

[1] P. W. Shor, Algorithms for quantum computation: discrete logarithms and factoring, in: Proceedings 35th annual symposium on foundations of computer science, IEEE, 1994, pp. 124–134.

[2] M. Grassl, B. Langenberg, M. Roetteler, R. Steinwandt, Applying grover's algorithm to aes: quantum resource estimates, in: Post-Quantum Cryptography, Springer, 2016, pp. 29–43.

[3] B. Blanchet, Security protocol verification: Symbolic and computational models, in: International Conference on Principles of Security and Trust, Springer, 2012, pp. 3–29.

[4] V. Cortier, S. Kremer, B. Warinschi, A survey of symbolic methods in computational analysis of cryptographic systems, Journal of Automated Reasoning 46 (2011) 225–259.

[5] M. Abadi, P. Rogaway, Reconciling two views of cryptography (the computational soundness of formal encryption), Journal of cryptology 15 (2002) 103–127.

[6] S.-L. Gazdag, S. Grundner-Culemann, T. Guggemos, T. Heider, D. Loebenberger, A formal analysis of IKEv2's post-quantum extension, in: Annual Computer Security Applications Conference, 2021, pp. 91–105.

[7] A. Hülsing, K.-C. Ning, P. Schwabe, F. Weber, P. R. Zimmermann, Post-quantum wireguard, in: 2021 IEEE Symposium on Security and Privacy (SP), IEEE, 2021, pp. 304–321.

[8] S. Escobar, C. Meadows, J. Meseguer, Maude-NPA: Cryptographic protocol analysis modulo

equational properties, in: Foundations of Security Analysis and Design V, Springer, 2009, pp. 1–50.
[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic, volume 4350, Springer, 2007.
[10] B. Blanchet, B. Smyth, V. Cheval, M. Sylvestre, Proverif 2.00: automatic cryptographic protocol verifier, user manual and tutorial, Version from (2018) 05–16.
[11] S. Meier, B. Schmidt, C. Cremers, D. Basin, The tamarin prover for the symbolic analysis of security protocols, in: International conference on computer aided verification, Springer, 2013, pp. 696–701.
[12] C. J. Cremers, The scyther tool: Verification, falsification, and analysis of security protocols, in: International conference on computer aided verification, Springer, 2008, pp. 414–418.
[13] J. Ramsdell, J. Guttman, CPSA4: A cryptographic protocol shapes analyzer, https://github.com/mitre/cpsaexp, 2018.
[14] V. Cheval, S. Kremer, I. Rakotonirina, The DEEPSEC prover, in: International Conference on Computer Aided Verification, Springer, 2018, pp. 28–36.
[15] D. D. Tran, K. Ogata, S. Escobar, S. Akleylek, A. Otmani, Formal specification and model checking of lattice-based key encapsulation mechanisms in Maude, in: Rewriting Logic and its Applications 14th International Workshop, WRLA 2022, 2022, p. 26.
[16] D. D. Tran, K. Ogata, S. Escobar, S. Akleylek, A. Otmani, Formal specification and model checking of saber lattice-based key encapsulation mechanism in Maude, in: Proceedings of the 34th International Conference on Software Engineering and Knowledge Engineering, 2022.
[17] D. Dolev, A. Yao, On the security of public key protocols, IEEE Transactions on information theory 29 (1983) 198–208.
[18] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1992) 73–155.
[19] M.-O. Stehr, J. Meseguer, P. C. Ölveczky, Rewriting logic as a unifying framework for petri nets, in: Unifying Petri Nets, Springer, 2001, pp. 250–303.
[20] N. Martí-Oliet, J. A. Verdejo-López, Implementing CCS in Maude, in: Actas de las VIII Jornadas de Concurrencia: Cuenca, 14 a 16 de junio de 2000, Universidad de Castilla-La Mancha, 2000, pp. 81–96.
[21] J. Meseguer, A logical theory of concurrent objects and its realization in the Maude language, in: G. Agha, P. Wegner, A. Yonezawa (Eds.), Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993, pp. 314–390.
[22] M. Katelman, S. Keller, J. Meseguer, Rewriting semantics of production rule sets, Journal of Logic and Algebraic Programming 81 (2012) 929–956.
[23] S. Liu, P. C. Ölveczky, J. Meseguer, Modeling and analyzing mobile ad hoc networks in Real-Time Maude, Journal of Logical and Algebraic Methods in Programming (2015).
[24] R. Bobba, J. Grov, I. Gupta, S. Liu, J. Meseguer, P. Ölveczky, S. Skeirik, Design, Formal Modeling, and Validation of Cloud Storage Systems using Maude, in: R. H. Campbell, C. A. Kamhoua, K. A. Kwiat (Eds.), Assured Cloud Computing, J. Wiley, 2018, pp. 10–48.
[25] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, Y.-M. Wang, A systematic approach to uncover security flaws in gui logic, in: 2007 IEEE Symposium on Security and Privacy (SP'07),

IEEE, 2007, pp. 71–85.
[26] J. Meseguer, G. Roşu, The rewriting logic semantics project, Theoretical Computer Science 373 (2007) 213–237.
[27] K. Bae, J. Meseguer, P. C. Ölveczky, Formal patterns for multirate distributed real-time systems, Science of Computer Programming 91 (2014) 3–44.
[28] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, K. Sonmez, Pathway logic: Symbolic analysis of biological signaling, in: Biocomputing 2002, World Scientific, 2001, pp. 400–412.
[29] C. Talcott, S. Eker, M. Knapp, P. Lincoln, K. Laderoute, Pathway logic modeling of protein functional domains in signal transduction, in: Biocomputing 2004, World Scientific, 2003, pp. 568–580.
[30] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, D. Stehlé, Crystals-kyber algorithm specifications and supporting documentation, NIST PQC Round 2 (2019).