UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dept. of Computer Systems and Computation

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

Master's Thesis

Master's Degree in Software Systems Engineering and Technology

AUTHOR: García Valero, Víctor

Tutor: Escobar Román, Santiago

Cotutor: Gutiérrez Gil, Raúl

Experimental director: SAPIÑA SANCHIS, JULIA

ACADEMIC YEAR: 2021/2022

ABSTRACT

Communication and information technologies shape the world's systems of today, and those systems shape the society we live in. The security of those systems relay on mathematical problems hard to solve for classical computers, that is, the available current computers. Recent advances in quantum computing threat the security of our systems and the communications we use. In order to face the threat, multiple solutions and protocols have been proposed. Kyber is one of them, specifically it is a key encapsulation mechanism that bases its security in the learning with errors problem over module lattices. This master's thesis 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 quantum adversaries suppose and study how Kyber works. We then construct in the system-specification language Maude, a symbolic model to represent the behaviour of Kyber in a network. On 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 with proving if a liveness and a security property hold.

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

RESUMEN

Las tecnologías de la información y la comunicación dan forma a los sistemas del mundo de hoy, y esos sistemas dan forma a la sociedad en la que vivimos. La seguridad de esos sistemas se basa en problemas matemáticos difíciles de resolver para las computadoras clásicas, es decir, las computadoras disponibles en la actualidad. Los avances recientes en computación cuántica amenazan la seguridad de nuestros sistemas y las comunicaciones que utilizamos con ellos. Para hacer frente a la amenaza se han propuesto múltiples soluciones y protocolos. Kyber es uno de ellos, en concreto se trata de un mecanismo de encapsulación de claves que basa su seguridad en el problema de aprendizaje con errores sobre retículos modulares. Este trabajo de fin de máster se centra en el análisis de Kyber para comprobar su seguridad bajo las asunciones del adversario del modelo de Dolev-Yao. Para conseguir este objetivo, primero aprendemos sobre el estado actual de las soluciones propuestas contra la amenaza que suponen los adversarios cuánticos y estudiamos cómo funciona Kyber. Luego construimos en el lenguaje de especificación del sistema Maude, un modelo simbólico para representar el comportamiento de Kyber en una red. En este modelo, llevamos a cabo un análisis de alcanzabilidad con el comando `search` y encontramos que hay un ataque *Man-In-The-Middle* presente. Luego, usamos la herramienta *Maude LTL logical model checker* para ampliar el análisis del sistema y probar si se mantienen unas propiedades de viveza y seguridad.

Palabras clave: Maude, logica de reescritura, verificación formal, protocolos post-cuánticos, mecanismos de encapsulación de claves

RESUM

Les tecnologies de la informació i la comunicació donen forma als sistemes del món d'avui, i aquests sistemes donen forma a la societat on vivim. La seguretat daquests sistemes es basa en problemes matemàtics difícils de resoldre per als ordinadors clàssics, és a dir, els ordinadors disponibles en l'actualitat. Els avanços recents en computació quàntica amenacen la seguretat dels nostres sistemes i les comunicacions que utilitzem amb ells. Per fer front a l'amenaça s'han proposat múltiples solucions i protocols. Kyber n'és un d'ells, en concret es tracta d'un mecanisme d'encapsulació de claus que basa la seguretat en el problema d'aprenentatge amb errors sobre reticles modulars. Aquest treball de fi de màster se centra en l'anàlisi de Kyber per comprovar-ne la seguretat sota les assumpcions de l'adversari del model de Dolev-Yao. Per aconseguir aquest objectiu primer aprenem sobre l'estat actual de les solucions proposades contra l'amenaça que suposen els adversaris quàntics i estudiem com funciona Kyber. Després construïm en el llenguatge d'especificació de sistememes Maude, un model simbòlic per representar el comportament de Kyber en una xarxa. En aquest model, portem a terme una anàlisi d'arribabilitat amb el comand `search` i trobem que hi ha un atac *Man-In-The-Middle* present. Després, fem servir l'eina *Maude LTL logical model checker* per ampliar l'anàlisi del sistema i provar si es mantenen unes propietats de vivesa i seguretat.

Paraules clau: Maude, lògica de reescritura, verificació formal, protocols post-quàntics, mecanismes d' encapsulació de claus

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2
- 1.3 Process . . . 3
- 1.4 Related work . . . 3
- 1.5 Structure . . . 4

2 Prior knowledge **6**
- 2.1 Dolev-Yao adversary model . . . 6
- 2.2 Maude . . . 7
  - 2.2.1 Sorts and subsorts . . . 8
  - 2.2.2 Operators . . . 8
  - 2.2.3 Axioms . . . 9
  - 2.2.4 Variables . . . 10
  - 2.2.5 System modules . . . 10
  - 2.2.6 Functional modules . . . 14

3 Key Encapsulation Mechanism KYBER **16**
- 3.1 Behaviour . . . 16
- 3.2 Security fundamentals . . . 17

4 Maude specification **19**
- 4.1 Assumptions . . . 19
- 4.2 Module composition . . . 19
  - 4.2.1 Functional modules . . . 20
  - 4.2.2 System module . . . 21
- 4.3 Anima . . . 27

5 Maude verification **30**
- 5.1 Reachability verification . . . 30
- 5.2 Formal verification . . . 31
  - 5.2.1 Predicates . . . 33
  - 5.2.2 Properties . . . 34
  - 5.2.3 Results . . . 34

6 Conclusion and future work **36**

List of Figures

1 Subtype relation between defined types for Vehicle, Car and Bike. 8
2 Definition of operators for types of bicycle and car. 9
3 Definition in Maude of a group operator with associative property and `empty` as the identity element in a group of vehicles. 10
4 Definition in Maude of the operator to represent the global state of the system consisting in two roads with two semaphores and groups of vehicles before and after the semaphores. 11
5 Rules that define the behavior of the system of an intersection with two traffic lights and definition of the initial state of our system. 12
6 Execution of the search command to search for a path between two terms in Maude. . 13
7 Run the search command to check the security of the junction. 14
8 Code of a functional module that implements the factorial in Maude. 14
9 Execution of two factorial expressions with the reduce command. 15
10 High level view of the behaviour of Kyber between two honest participants, id est, Alice and Bob. 16
11 Algorithms of Kyber Key Encapsulation Mechanism. 17
12 Internal algorithms of the Kyber Key Encapsulation Mechanism. 18
13 Definition of the property to cancel the noise present in the component of a ciphered text *c* in functional module *ENCRYPTION*. 21
14 Definition of a participant at the network in our system module KYBERV2. 22
15 Definition of a message at the network in our system module KYBERV2. 23
16 Definition of the global state of the system in our system module KYBERV2. 23
17 Definition of conditional rule *KeyGen* in our system module KYBERV2. 23
18 Definition of conditional rule *SendPK* in our system module KYBERV2. 24
19 Definition of conditional rule *RecievePK* in our system module KYBERV2. 24
20 Definition of conditional rule *Enc* in our system module KYBERV2. 25
21 Definition of conditional rule *SendCiph* in our system module KYBERV2. 25
22 Definition of conditional rule *RecieveCiph* in our system module KYBERV2. 25
23 Definition of conditional rule *Dec* in our system module KYBERV2. 26
24 Definition of a rule *Comp* in our system module KYBERV2. 26
25 Definition of conditional rule *Intercept1* in our system module KYBERV2. 27
26 Definition of conditional rule *Intercept2* in our system module KYBERV2. 27
27 Execution trace of our symbolic model from *init* state till step *Enc* showed by the online web tool Anima. 28
28 Execution trace of our symbolic model from *SendCiph* till step *ReceieveCiph* showed by the online web tool Anima. 28
29 Execution trace of our symbolic model on *Dec* step showed by the online web tool Anima. 29
30 Declaration and definition of two initial states for our system module KYBERV2 in Maude. 30
31 Command template for a man-in-the-middle attack search in Maude. 31
32 Results on the reachability analysis over the first initial state in module KYBERV2. . 31
33 Results on the reachability analysis over the second initial state in module KYBERV2. 32
34 Declaration of three predicates in KYBERV2-PREDS. 33

35 Definition of predicate *wantsToShareKey* in KYBERV2-PREDS. 33
36 Definition of predicate *sharedAKeyWith* in KYBERV2-PREDS. 33
37 Definition of predicate *stolenSecret* in KYBERV2-PREDS. 34
38 Definition of a property about liveness for our symbolic model KYBERV2. 34
39 Definition of a property for security for our symbolic model KYBERV2. 34
40 Results on the LTL satisfability over the liveness property over the symbolic model for Kyber. 35
41 Results on the LTL satisfability over the security property over the symbolic model for Kyber. 35

Chapter 1

1 Introduction

We provide some subsections in order to introduce the work exposed in this master's thesis. The sections of our introduction are as follows. In the first place we will take a look at the motivations that led to the development of this work in the field of formal methods. The next subsection introduces and sets the objectives we aim to achieve at the end of the master's thesis. Then, in the following subsection, we explain the procedure followed to fullfill the previous defined objectives. The following subsection exposes the research done on related work available from the literature, identifying papers and tools of interest. Last subsection gives an overlook on the structure of the rest of this master's thesis giving, for each section, a brief explanation.

1.1 Motivation

Society is nowadays deeply connected thanks to the last decades advances on communication technologies. It also has all the information at the reach of their hands thanks to information technologies. Such capabilities of communication between agents and access to any kind of information, be it sensitive, private or even public, comes with a cost. The cost is a structure or architecture built over the years with different versions of protocols and software, that must work together, and as any structure with important purposes, must be secure. Thus, security is keystone for the current foundations of society.

Today's security relays heavily on complexity problems. Most of the current network infrastructure and systems work over classical computers. Specifically, most of these protocols relay on three types of problems that are 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 of category NP, which stands for non polynomial time, for classic computers. The meaning of the category NP can be viewed as that: Such problems are considered to be unsolvable by classic computers in a time where the information gained is worth the time it takes to solve the problem.

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

In order to face the threat quantum computers suppose to the security of society's infrastructure the National Institute for Standards and Technologies (NIST) started on 2017 the Post-Quantum Cryptography Project (PQC). The project is carried as a competition, divided in multiple rounds, in order to analyze candidate protocols to be used in an standard as a solution against quantum adversaries. At present there has been 4 rounds on the project and the candidates vary from public-key encryption and key-establishment algorithms to digital signature algorithms. We select round 3, which finalized

on 2020, because round 4 was announced on the 5th of July on 2022. It is important to remark that in round 4, which marks the near end of the pryetc, among the selected protocols is Kyber, the only public-key encryption and key-establishment algorithm, meaning it will likely be made standard by NIST. The third-round finalist on public-key encryption and key-establishment algorithms are Classic McEliece [3], CRYSTALS-KYBER [4], NTRU [5], and SABER [6]. Third-round finalists for digital signatures are CRYSTALS-DILITHIUM [7], FALCON [8], and Rainbow [9]. In this master's thesis we focus 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 in the hardness of solving the Learning With Errors problem over module-latices. These KEM main goal is to securely share a given key between two participants of a network where channels are not safe from intruders. Such goal is interesting for conventional cryptography, also known as Symmetric Cryptosystem, which uses a secret key to encrypt a message. The same key is required to decrypt the encrypted message. Therefore, the sender and the receiver of an encrypted message either had to agree on the secret key before encrypting the message, or the receiver had to learn the secret key before decrypting it. The same secret key could not be used when sending a message to a different receiver. 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 there are two kinds of approaches that can be taken: computational security and symbolic security. The former is based on mathematical proofs on a computational model, where messages are bit strings and the adversary is any probabilistic Turing machine. Length of keys are determined by a value called security parameter, and the adversary runtime is supposed to be polynomial under such security parameter. This is the approach generally used by cryptographers and the authors of Kyber have already covered this approach. The later is based on the use of symbols, where the cryptography primitives are function symbols acting as black boxes. Messages of symbolic models are terms on these primitives and the adversary is restricted to use the defined primitives. It is important to note that these models assume perfect cryptography, i.e., cipher-texts cannot be broken without the proper key.

Despite the fact that 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 master's thesis, our experiments belong to the later. That is why this master's thesis follows the symbolic security approach. It is important to mention that this approach not only can be applied to the selected protocol but also to any other scheme or mechanism on rounds 3 or 4 of the Post-Quantum Cryptography project.

1.2 Objectives

The main goal of this master's thesis is to analyze, from a symbolic security point of view, a post-quantum protocol and verify properties about it. Our main goal can be divided in sub goals for better tracing as follows.

- Search for state of the art post-quantum protocols
- Study the principles and basic procedure of the protocol

- Construct a symbolic model to represent the behaviour of participants
- Add to the model the behaviour of an adversary under Dolev-Yao intruder model
- Specify properties in two ways:
  - Properties that say the model works as the official specification states
  - Properties that say if a MITM attack is found
- Verify those properties on the model with reachability analysis and the application of the logical model checker of Maude.

1.3 Process

The methodology followed to develop the proposed solution is based in 6 incremental steps, each one is part of the protocol and its the environment that we try to represent. The following is a list of these steps and its main objective during model construction.

1. Messages: Model the basic interaction on the network between participants, content of the messages is irrelevant.
2. Keys: Define types of keys and how are they managed and stored by participants.
3. Matrices and vectors: How to portray the mathematical foundations of the protocol in the elements of our model such as keys.
4. Equality: Construct the necessary equational theory to specify the protocol in a functional language.
5. Intruder: Include the capabilities of an adversary under Dolev-Yao's adversary model.
6. Search of MITM attack: Conduct a search to verify if a Man In The Middle attack is present in the constructed model.

It is important to state that only the resulting symbolic model from the final step 6 is shown in Section 4.

1.4 Related work

Current advances in security of protocol analysis have been made. One interesting idea is the one proposed at [10] where the author explains several examples on the formal specification of protocols, where the symbolic and computational model analysis are introduced and explained. We also found [11], a survey where the authors explore the current literature and papers on both symbolic and computational analysis of protocols. In this survey they analyze the result of combining both types of analysis, a proposal that was made initially by [12] in order to close the gap between both lines of protocol verification. One can also find specific papers [13][14] applying symbolic, computational or both analysis 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 it. In the later, 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 a key-encapsulation mechanisms (KEMs). The authors verify their proposal's security with symbolic and computational proofs. On 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 protocol analysis tools we have Maude-NPA [15], related to the programming language Maude [16]. Maude-NPA has a theoretical basis on rewriting logic, unification and narrowing, and performs backwards search from a final attack state to determine whether or not it is reachable from an initial state. Other tools such as ProVerif [17] are based on an abstract representation of a protocol using Horn clauses and the verification of security properties is done by reasoning on these representative clauses. Another tools such as Tamarin [18] are based on constraint solving to perform an exhaustive, symbolic search for executions traces. And other tools such as Scyther [19] or CPSA [20] attempt to enumerate all the essential parts of the different possible executions of a protocol. We also want to point out that we are not aware of any works using these tools for the purpose of verification of cryptography post-quantum protocols.

Finally, there is a first symbolic security analysis of a collection of post-quantum protocols, among which Kyber is found, in [21] using Maude. For each of those protocols a man in the middle (MITM) attack is found. In this master's thesis we aim to provide a higher level or more abstract model of Kyber and a symbolic analysis than [21]. A more abstract model is needed in order to better understand and reason on KEMs for non experts of cryptography. The symbolic analysis aims to demonstrate not only that a MITM attack is present, but prove that our model considers all protocol's possible behaviours. In [22], the same authors have used the methodology from [21] for the KEM known as SABER, a close relative of Kyber. It may be of interest to us in applying our new methodology proposed in this master's thesis to other key encapsulation mechanisms like SABER.

1.5 Structure

The manuscript is structured in the following chapters. We provide a brief explanation of the contents of each chapter to lay out the reader an idea of what to expect.

- **Chapter 2:** Explains key concepts about cryptography and also introduces the Dolev-Yao adversary model. Then provides a detailed explanation of the Language Maude and introduce a simple example for comprehension purposes.

- **Chapter 3:** Provides a detailed description in different levels of the Key Encapsulation Mechanism KYBER, explaining its behaviour and hardness. It also explains the security basis of the protocol against quantum adversaries.

- **Chapter 4:** Lists the assumptions taken over the protocol, both in a mathematical and specification way. Then explains the proposed solution and an overview of its behaviour is shown using the web tool ANIMA.

- **Chapter 5:** Presents two kinds of verification, reach-ability analysis with *search* command and formal verification of some LTL properties with model checking.

- **Chapter 6:** Provides the conclusion of this master's thesis, checking the achieved objectives and discussing the results obtained. This chapter also proposes future directions on the line of research for the work done.

Chapter 2

2 Prior knowledge

In an effort to understand subsequent chapters we introduce now 'key' concepts necessary to understand the work done. For this purpose, first, we describe a model for representing an adversary in a communication network. We then explain the basic fundamentals and notions for term rewriting, and present an introduction to the language Maude.

2.1 Dolev-Yao adversary model

The Dolev-Yao adversary model was introduced in [23]. In this paper authors explained that public key schemes are secure against adversaries that can not modify the environment, which is not realistic. 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 type of intruder can only read the message and extract from them raw content, that means they can not derive any contents from messages without the proper private information. The later type of intruder can not only read messages but also modify them and send them through the network. It is important to clarify that the intruder is considered to be a polynomial time Turing machine.

In this seminal work, 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 of the 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 participants of the network, 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 can not be changed and must be followed by any participant of it, that is, a user can not do undefined steps during the protocol execution.
- Everyone can encrypt with the public key of a participant.
- Only a participant knows how to decrypt the content of a message that was encrypted with his public key.

2.2 Maude

Maude [16] is based on rewriting logic, 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 [24] to a process calculus [25], from an object-based system [26] to asynchronous hardware [27], from a mobile ad hoc network protocol [28] to a cloud-based storage system [29], from a web browser [30] to a programming language with threads [31], or from a distributed control system [32] to a model of mammalian cell pathways [33, 34]. And all *without any encoding*: what you see and get is a direct definition of the system itself, without any artificial encoding.

Maude is based on rewriting logic but rewriting logic has a sub-logic, called membership equational logic, for those deterministic or functional parts of a system. That is, an equational program is a functional program in which a functional expression (called a term) is evaluated using the equations in the program as left-to-right rewrite rules, which are assumed confluent and terminating. If such an evaluation terminates, it returns a unique computed value (determinism), namely, its normal form after simplifying it with the (oriented) equations.

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, for example terms in an equivalence class associated to cryptography properties. We can call this aspect the static part of the specification. Instead, its dynamics, i.e., how states evolve, is 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 in a given state several transition rules may be applied concurrently to different sub-parts, producing several concurrent local state changes, and that rewriting logic itself models those concurrent transitions as logical deductions [35].

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 both verify invariants or to 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, and this will be uncovered by the above `search` command since all reachable sates are explored in a breadth-first manner. If, instead, 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. But if there is an infinite number of states reachable from the initial state, search will never terminate. A standard approach is to bound the analysis, e.g. by specifying a depth bound for the search command. Another approach is to perform an over or under approximation of the search space, so that the system becomes finite-state and the invariant can be verified.

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.

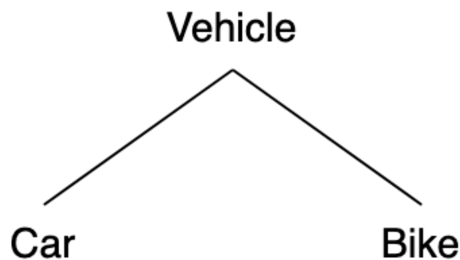

Figure 1: Subtype relation between defined types for Vehicle, Car and Bike.

2.2.1 Sorts and subsorts

At Maude, the first thing we do when creating a module, whether it is functional or a system module, is define the types that will be used in the operators and variables that we want to use. To define a type we must use the reserved word `sort` followed by an identifier. An identifier is a string of unreserved characters that Maude uses to identify everything from a module name to a variable name, or in this case, a type.

Let's see an example of how to declare sorts in order to introduce the concept of subtype. The statement "`sort Car .`" defines a type with identifier Car. The use of the dot in Maude is to terminate statements, similar to semicolons in other languages such as Java, Python or C. If we want to define several types at the same time we can use the `sorts` keyword. Thus, the statement "`sorts Car Vehicle Bike .`" declares the types *Car, Vehicle and Bike*.

Then we have the concept of *subsort*, which is a relationship between different types so that a type A that is a subtype of a type B is also a type B. To clarify this concept we will use the types that we have already defined before *Car, Vehicle and Bike*. To declare a relationship of subtypes there is in Maude the reserved word `subsort`. Therefore, if we want to declare that the type *Car* is a subtype of *Vehicle*, we write the statement "`subsort Car < Vehicle .`" where the symbol < is the operator reserved for indicating the direction of the relationship. We can do the same for Bike with "`subsort Bike < Vehicle .`".

With the previous statements, we obtain a relationship that resembles the inheritance of other languages. As we see in Fig. 1, we have the three types with Car, Bicycle and Vehicle in a tree like structure, being Car and Bicycle subtypes of Vehicle.

2.2.2 Operators

The operators allow us to define the syntax that the types that we have defined will have, that is, the notation that will express what we need. In this way we can model the elements of a system or function with great freedom of expression.

Operators in Maude are defined by the reserved word `op` followed by the definition of the notation. Operators can be parameterized using the underscore character to indicate the position of each of the parameters within the notation. We can also parameterize them if we omit any underscore characters, making it a prefix operator, that is, we will pass the parameters using parentheses as if it were a Java or C function. Once the parameters are set, we will indicate, after a colon, the types of each parameter

```
*** Definition of types for bike (mo : Mountain, ci : City)
ops mo ci : -> BikeType .

*** Definition of a car with a given plate number
op <c(_)> : Nat -> Car .

*** Definition of a bike with its number id and type
op <b(_)t[_]> : Nat BikeType -> Bike .
```

Figure 2: Definition of operators for types of bicycle and car.

in order of appearance and finally the type returned by the operator after an arrow.

If we continue with the previous example of types of Vehicle, we can define an operator for each type of vehicle, each having an identification number as a parameter, which may well be its license plate in the case of a car or a serial number in the case of a car. to be a bicycle. Also, for bicycles we can define a *BicycleType* type with notations for mountain and city types. We make use of Maude and get the following code snippet in Fig. 2.

In Fig. 2 we have the first statement for the definition of bicycle types as `mo` and `ci`, which will later be used in the definition of the notation for the bicycles in the last statement. Therefore, a bicycle with serial number one and a city type can be written as `<b(1)t[ci]>`. For the car we have the intermediate sentence which receives a natural number as a license plate.

2.2.3 Axioms

Axioms are properties or attributes on sets that are satisfied on a certain operator when it is defined in a sequence of another operator. These attributes are applied in Maude if we write them between the symbols `[` and `]` at the end of the sequence operator, that is, the operator that contains a sequence of other operators of the same type. The axioms defined by Maude are:

- `assoc` for specifying associativity property on the set
- `comm` for specifying commutativity property on the set
- `id:e` for specifying the identity element `e` for the set

To see an example of how to apply attributes, we are going to use the Vehicle type to define an operator that indicates a set of vehicles on which we will apply properties of interest. We first create the type *GroupOfVehicles*, for example, using the statement "`sort GroupOfVehicles .`". Later we indicate through a relation of sub-types, for example "`subsort Vehicle < GroupOfVehicles .`", that a vehicle is a type of set of vehicles, that is, a set where there is only one element. Fig. 3 below shows the code needed to create a set of vehicles in Maude with all three set properties.

In the second statement we define the `empty` operator that will act as the identity element for the set of vehicles. Then in the third statement we define the operator for a set of vehicles as a sequence of other sets of vehicles. Take a look at how we finally define the two attributes for the new operator, indicating the associative property and the `empty` identity element.

```
*** Relation declarations between sorts
subsort Vehicle < GroupOfVehicles .

*** Definition of the identity element for groups of vehicles
op empty : -> GroupOfVehicles .

*** Definition of a group of vehicles with an order
op __ : GroupOfVehicles GroupOfVehicles -> GroupOfVehicles [assoc
id: empty] .
eq V:Vehicle V:Vehicle = V:Vehicle .
```

Figure 3: Definition in Maude of a group operator with associative property and `empty` as the identity element in a group of vehicles.

It is interesting to look at the last sentence where we have defined an equation to indicate that, if there are two identical vehicles in a set of vehicles, one of them is eliminated from the set. In this way, we define in Maude the elimination of duplicates on sets.

2.2.4 Variables

Variables in Maude are defined using the structure "`var <id> : <var-sort> .`", where `<id>` is the name by which to identify the variable and `<var-sort>` the type it can contain. If we want to define more than one variable of the same type in a single statement, we can replace the previous `var` with `vars`.

As an example we can define three variables with identifiers *c*,*b*, *v*. The identifier *c* is for a variable of type Car. For *b* the type will be Bike. While for *v* it will be Vehicle as its type. An operator can be stored in each variable that returns the type specified for the variable. Furthermore, thanks to the sub-type relationship, the variable *v* can also store the contents of *c* or *b*.

2.2.5 System modules

System modules allow us to model systems in Maude, having non deterministic behavior with the possibility of infinite executions. We define them using the structure `mod <ModuleName> is <DeclarationsAndStatements> endm`, where `<ModuleName>` is the module identifier in uppercase and `<DeclarationsAndStatements>` are statements. As statements we can make use of those that we have already seen, such as types and operators, although there are also equations and rules.

System modules specify rewrite theories. Theories are described in the form R = ($\sum$, E, φ, R). The element $\sum$ represents the set of all states of the system. The symbol E represents the equational theories defined by the equations of the module. The symbol φ is a function that assigns a set of natural numbers to the states of $\sum$ according to their number of arguments. The last element, R, is the set of system rules that specify the transitions between the states in $\sum$.

Equations are statements of the form "`eq <Term-1> = <Term-2> .`", and they transform the left part of the equal to the right part, so that `Term-1` becomes `Term-2`. One way to view equations is as deterministic transformations that allow us to reduce or simplify expressions.

```
*** Definition of a juntion (our system) with 2 semaphores and
*** multiple groups of vehicles that want to cross
op cardo [_] (_) [_] | decumano [_] (_) [_] : GroupOfVehicles
Semaphore GroupOfVehicles GroupOfVehicles Semaphore
GroupOfVehicles -> JunctionWith2Semaphores .
```

Figure 4: Definition in Maude of the operator to represent the global state of the system consisting in two roads with two semaphores and groups of vehicles before and after the semaphores.

The rules are like the equations, except that by applying them we are changing the state within the modeled system, that is, there is a relative increase in time. Rules have the form "`rl [id] : l => r .`", having an optional identifier `id` and converting the left part `l` to the right part `r`. Let's see Fig. 5, where an example to explain the system modules that what we have already seen of cars and bicycles is presented.

Suppose we want to model the behaviour of a crossroads with two traffic lights, where there are two types of vehicles present, cars and bicycles, for example. At this intersection we will call the north-south road *cardo* and the east-west road *decumano*. The possible states of the traffic lights will be green or red. At the intersections there may be vehicles, of which we will have cars and bicycles as we have defined them in subsection 2.2.2.

We now define what the representation of the system is as an operator, for which we have defined the type JunctionWith2Semaphores. This global state operator is defined by the statement in the following figure.

The sentence tells us that the roads are separated by the symbol `|` and they are identified with the words `cardo` and `decumano` at the beginning. The parameters on each road are two sets of vehicles in brackets and a traffic light between the sets of vehicles in parentheses.

Once we have the representation of the system, we can give behavior thanks to the rules. To simplify the model we are going to make it so that vehicles can only cross in a straight line and in one direction to make the model terminate. Fig 5 shows the rules defined, four for crossing vehicles and two for changing the state of the traffic light.

In the first place, we define the variables that we will need in the rules. We have defined four variables `CONJV1`, `CONJV2`, `CONJV3` and `CONJV4` as sets of vehicles, and a variable `veh` variable of type `Vehicle` to be able to represent both cars and bicycles.

Then we look at the first two crossing rules, with identifiers `CrossCardo` and `CrossDecumano`, where a vehicle can cross when the corresponding traffic light on his road is green. The same principle is applied for the rules with identifiers `CrossCardo2` and `CrossDecumano2`, but in this case it is applied when there is only one vehicle left, making the set remain empty once it crosses.

Finally, we define two rules to represent the change of state of the traffic lights. Both rules cause the states of the two semaphores to be exchanged with each other.

In relation to the rules, it is worth noting the existence of conditional rules. Conditional rules are similar to normal rules except they have the syntax "`crl [<label>] : <Term-1> => <Term-2> if <Condition-1> /\ ... /\<Condition- k> .`" where the statements "`<Condition-1> /\ ... /\ <Condition-k>`" are expressions that will return false or true. Conditions in conditional rules can take three forms:

1. Equations with the syntax `t=t'`.

```
*** Variables
vars CONJV1 CONJV2 CONJV3 CONJV4 : GroupOfVehicles .
var VEH : Vehicle .
vars SEM1 SEM2 : Semaphore .

*** System rules
rl [CrossCardo] : cardo [VEH CONJV1] (green) [CONJV2] | decumano
[CONJV3] (red) [CONJV4] => cardo [CONJV1] (green) [VEH CONJV2] |
decumano [CONJV3] (red) [CONJV4] .
rl [CrossDecumano] : cardo [CONJV1] (red) [CONJV2] | decumano
[VEH CONJV3] (green) [CONJV4] => cardo [CONJV1] (red) [CONJV2] |
decumano [CONJV3] (green) [VEH CONJV4] .

rl [CrossCardo2] : cardo [VEH] (green) [CONJV2] | decumano
[CONJV3] (red) [CONJV4] => cardo [empty] (green) [VEH CONJV2] |
decumano [CONJV3] (red) [CONJV4] .
rl [CrossDecumano2] : cardo [CONJV1] (red) [CONJV2] | decumano
[VEH] (green) [CONJV4] => cardo [CONJV1] (red) [CONJV2] | decumano
[empty] (green) [VEH CONJV4] .

rl [ChangeSemaphores1] : cardo [CONJV1] (red) [CONJV2] | decumano
[CONJV3] (green) [CONJV4] => cardo [CONJV1] (green) [CONJV2] |
decumano [CONJV3] (red) [CONJV4] .
rl [ChangeSemaphores2] : cardo [CONJV1] (green) [CONJV2] |
decumano [CONJV3] (red) [CONJV4] => cardo [CONJV1] (red)
[CONJV2] | decumano [CONJV3] (green) [CONJV4] .

op init : -> JunctionWith2Semaphores .
eq init = cardo [<c(1)> <b(1)t[mo]>] (red) [empty] | decumano
[<c(2)>] (green) [empty] .
```

Figure 5: Rules that define the behavior of the system of an intersection with two traffic lights and definition of the initial state of our system.

```
search in CROSS-JUNCTION : init =>* cardo[empty](SEM1)[<b(1)t[mo]> <c(1)>]|
    decumano[<c(2)>](SEM2)[empty] .

Solution 1 (state 6)
states: 7  rewrites: 10 in 0ms cpu (0ms real) (222222 rewrites/second)
SEM1 --> green
SEM2 --> red

Solution 2 (state 10)
states: 11  rewrites: 16 in 0ms cpu (0ms real) (213333 rewrites/second)
SEM1 --> red
SEM2 --> green

No more solutions.
states: 12  rewrites: 25 in 0ms cpu (0ms real) (240384 rewrites/second)
```

Figure 6: Execution of the search command to search for a path between two terms in Maude.

2. Matching equations with the syntax `t:=t'`.

3. Rewrite expressions with the sintax `t->t'`.

As a command of interest in the system modules we have `search`, allowing us to explore for paths from one state to another that we indicate. This command allows the use of different symbols to tell Maude the criteria that must meet to stop the search. By default in Maude, searches are performed on a state graph using the breadth first search approach.

The `search` command has the form "`search <Term-1> <SearchArrow> <Term-2> .`", where `<Term-1>` and `<Term-2>` are valid state statements of our modeled system. The word `<SearchArrow>` can be one of the following forms, each one giving a different criterion to proceed in the construction of the state graph:

- `=>1` to make a search of a single execution step, that is, only one rule is applied.
- `=>+` to make a search of one or more rewrite rules.
- `=>*` to make a search of zero, one or more rewrite rules.
- `=>!` to make a search of only canonical states, in other words, a state to which no rules can be applied leading to new states not explored before.

For example, for the system module of the crossing with two traffic lights, we can query if there is a path from a state with several cars in the *cardo*, to one where all the cars in the *cardo* have crossed the traffic light. We can add all the specifications that we want as long as we respect the defined syntax, that is, that we build the states as we have specified them in the system module. Let's look at some examples of searches.

As a first example we are going to write a command that finds us if it is possible to get from a state with two vehicles in the *cardo*, for example, a car and a bicycle, and a car in the *decumanus*, to a state with the two vehicles in the *cardo*, but no vehicle in the *decumanus*. The sentence is written as shown in Fig. 6, obtaining a solution from Maude by applying four rules.

As a second example, we can try something more interesting. We are going to see if, in the system we have modeled, the two traffic lights can turn green simultaneously. The searches for undesired

```
search in CROSS-JUNCTION : init =>* cardo[CONJV1](green)[CONJV2]| decumano[
    CONJV3](green)[CONJV4] .

No solution.
states: 12  rewrites: 25 in 0ms cpu (0ms real) (134408 rewrites/second)
```

Figure 7: Run the search command to check the security of the junction.

```
fmod FACT is
    protecting NAT .
    op _! : Nat -> Nat .
    var N : Nat .
    --- factorial for N=0 is 1
    eq 0 ! = 1 .
    --- factorial for N>0 is (N-1)! * N
    eq N ! = (sd(N,1))! * N [owise] .
endfm
```

Figure 8: Code of a functional module that implements the factorial in Maude.

states will not be successful when Maude does not find any solution, since it will mean that from the specified initial state the situation that we consider critical in the system is not reached.

Fig. 7 shows the use of search with the same initial state as before but now it has a final state where *cardo* and *decumano* have green traffic lights. This is an undesirable situation that should not occur at a junction because it could cause an accident for the users of the system.

The other path to formally specify and verify properties on system modules is by using the model checking technique. Maude has a model checker [36] to perform model checking by writing properties as formulas in linear temporal logic and using modules to describe predicates about the states of the system. We further explain this concept and put it into practice in Section 5.

2.2.6 Functional modules

Functional modules specify functions and are similar to system modules except that they can only contain equations. Functional modules, unlike system modules, are deterministic and finite. To define a functional module in Maude, we do so between the `fmod` and `endfm` keywords. Let's see an example of specifying a functional type module.

If we look at the following figure, we have a functional module, defined between `fmod` and `endfm`, which specifies through equations the factorial mathematical function of a number *N*. In the module we have defined the symbol for said operation and the equations that will transform the expression until it is irreducible, and thus obtain the result.

Continuing with Fig. 8, on line three we specify the symbol of the factorial function, which obtains a natural number in place of the _ symbol and returns another natural number, as indicated by the expression `Nat -> Nat`.

Then on line four we declare the variable `N`, which we will use in the following equations on lines five and six. With the first equation we define that when Maude finds a `"0 !"` on the left side this is translated into the number one. The second equation will be executed whenever the first one is not

```
Maude> reduce 3 ! .
reduce in FACT : 3 ! .
rewrites: 10 in 0ms cpu (0ms real) (46082 rewrites/second)
result NzNat: 6
Maude> reduce 0 ! .
reduce in FACT : 0 ! .
rewrites: 1 in 0ms cpu (0ms real) (1000000 rewrites/second)
result NzNat: 1
```

Figure 9: Execution of two factorial expressions with the reduce command.

executed thanks to the attribute defined *owise*, and it will return the factorial function of a number *N* as a factorial of *N-1* multiplied by *N*.

In addition, it should be noted that just as there are conditional rules, there are also conditional equations. These behave in the same way as the equations described with the difference that one or more conditions must pass in order to be applied, in the same way as the conditions of the rules. The syntax for conditional equations is "`ceq <Term1> = <Term2> if <Condition-1> / ... / <Condition-k> .`", where conditions `<Condition-k>` are expressions that will return false or true. The conditions of the conditional equations can take two forms:

1. Be other equations, with the form `t = t'`.
2. Matching equations, with the form `t := t'`.

To finish, we are going to see the `reduce` command. This command is very useful for us to test equations that we have defined, either in functional modules or in system modules. The reduce command receives an expression and returns the maximum reduced expression using the equations we have defined.

As an example we will use the definition of the factorial operation in the *FACT* functional module to test different expressions. For example, as Fig. 9 shows, we tried to reduce factorial of three and factorial of zero. The first returns six and the second returns one, both of which are correct. In addition, since no rules are applied in reduce and there are no rules in functional modules, we can see how in both executions the number of rewrites is zero, indicating that no rule was applied and therefore the state was not changed.

Chapter 3

3 Key Encapsulation Mechanism KYBER

In this chapter we will explain the selected protocol, Kyber. For this purpose first we explain the behaviour from a high level point of view. Then we elaborate on how is Kyber quantum resistant by explaining mathematical concepts while we deep down into the low level code of Kyber, remarking the important components that provide Kyber to resist a quantum adversary.

3.1 Behaviour

In order to explain the behaviour of Kyber we should look at Fig. 10. Participants of the protocol will be Alice and Bob for literature reasons, but it should be noted that this is only a formalism and both represent any honest user of the network that want to establish a shared key with another user of the same network.

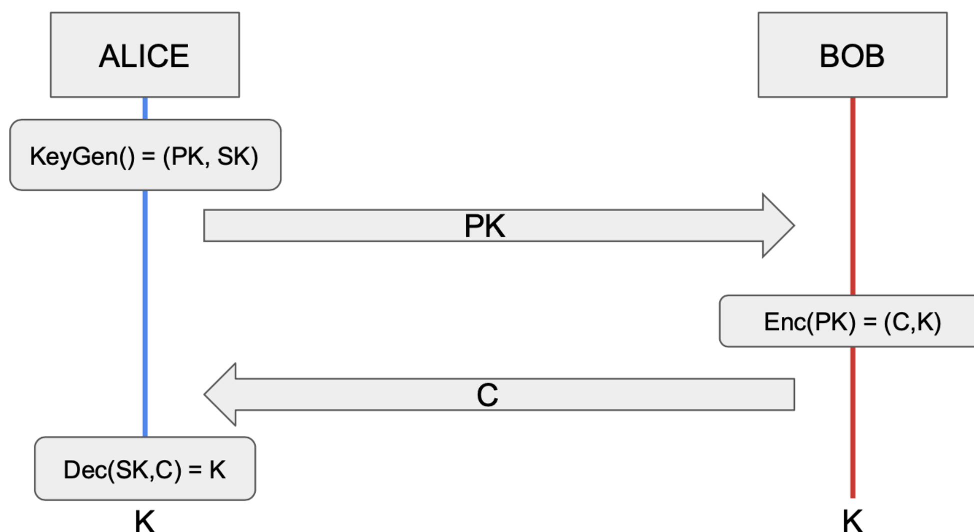

Figure 10: High level view of the behaviour of Kyber between two honest participants, id est, Alice and Bob.

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 participant of the network. The later, *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, namely Bob, with her the public key. Once Bob receives the public key of Alice, he performs *Enc*, which stands for *Encrypt*, using the received public key. Function *Enc* provides 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 second one in a new message back to Alice, the one that started the protocol and from which 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*. Later we will see why we state 'ideally' for the decryption of a ciphered text. With the last step, both participants have shared a key *K* between them in a secure way.

$$\begin{aligned} Alice &\rightarrow Bob : pk \\ Alice &\leftarrow Bob : c \end{aligned} \tag{1}$$

As we have seen the network is very simple and the interaction between participants is minimal. Eq. 1 depicts there are only two messages flowing through the network during protocol execution. No confirmation messages or any prior establishment to know where is the participant in the network is performed. Such discovery and set-up work is assumed to have been done previously.

$$\begin{aligned} KeyGen() &\rightarrow (pk, sk) \\ Enc(pk) &\rightarrow c \\ Dec(c) &\rightarrow k \end{aligned} \tag{2}$$

Finally, about the cryptography primitives used and depicted in Eq. 2 we must remark there are two versions for each of them as we will see in the next section. For this previous explanation on the behaviour we only need to remember these functions as black boxes with an input and an output.

3.2 Security fundamentals

Let us now see why these primitives, that 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 on them, such as concatenation, transposition, product or other more complex ones such as hash and key derivation functions. These operations can be seen in Fig. 12 and are present in the three main functions.

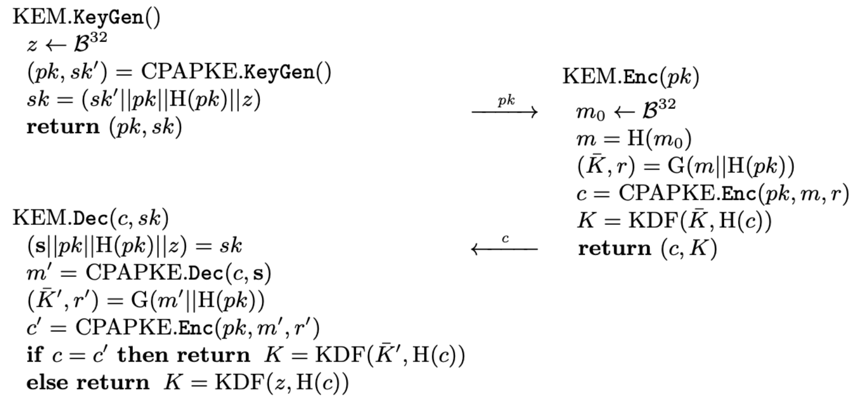

Figure 11: Algorithms of Kyber Key Encapsulation Mechanism.

It should be noted, here in Fig. 11, that in *KYBER.Dec(c,sk)* when the sub-function *CPAPKE.Dec(c,s)* occurs, the computed text *m'* could not 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 it's 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 occurs with low probability, but it states that encryption and decryption phases are not error prone.

But 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 showed in Fig. 12, we can see in algorithm *CPAPKE.Enc* there are vector values such as e_1 and e_2 that have been sampled with a random seed *r* using function *sampleCBD* from a centered binomial distribution (*CBD*). 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.

$$X' = Decompress(Compress(X, 1), 1) \tag{3}$$

The error is tried to be eliminated by function *Decompress* when extracting from the pair of vectors c_1 and c_2 the vectors *u* and *v* respectively, at the middle of step *CPAPKE.Dec*. This function has a property in combination with function *Compress*, which states in Eq. 3 that decompressing the compress of a given value *X* with same second parameters, it gives a new value *X'* which is similar to the original compressed value. This property takes place while computing *Compress(v'-*s^T*u',1)*.

CPAPKE.`KeyGen`()
$d \leftarrow \mathcal{B}^{32}$
$(\rho, \sigma) = \mathrm{G}(d)$
$R_q^{k \times k} \ni \mathbf{A} = \mathrm{generate}(\rho)$
$R_q^k \ni \mathbf{s}, \mathbf{e} \leftarrow \mathrm{sampleCBD}(\sigma)$
$\mathbf{t} = \mathbf{As} + \mathbf{e}$
$pk = (\mathbf{t} || \rho)$
$sk = \mathbf{s}$
return (pk, sk)

CPAPKE.`Dec`(c, sk)
$(c_1 || c_2) = c$
$\mathbf{u}' = \mathrm{Decompress}_q(c_1, d_u)$
$v' = \mathrm{Decompress}_q(c_2, d_v)$
$m' = \mathrm{Compress}_q(v' - \mathbf{s}^T \mathbf{u}', 1)$
return m'

CPAPKE.`Enc`(pk, m, r)
$(\mathbf{t} || \rho) = pk$
$R_q^{k \times k} \ni \mathbf{A} = \mathrm{generate}(\rho)$
$R_q^k \ni \mathbf{r}, \mathbf{e}_1 \leftarrow \mathrm{sampleCBD}(r)$
$R_q \ni e_2 \leftarrow \mathrm{sampleCBD}(r)$
$\mathbf{u} = \mathbf{A}^T \mathbf{r} + \mathbf{e}_1$
$v = \mathbf{t}^T \mathbf{r} + e_2 + \mathrm{Decompress}_q(m, 1)$
$c_1 = \mathrm{Compress}_q(\mathbf{u}, d_u)$
$c_2 = \mathrm{Compress}_q(\mathbf{v}, d_v)$
return $c = (c_1 || c_2)$

Figure 12: Internal algorithms of the Kyber Key Encapsulation Mechanism.

It is important to mention that other operations take place such as *generate*, *sampleCBD*, *Compress*, *Decompress*, *encode* and *decode*. These are operations necessary for the main three operations we have described before but are not explained in detail in this master's thesis because they are not necessary for our understanding of the protocol. Full descriptions of these specific functions are available at [4].

Chapter 4

4 Maude specification

This chapter is divided in three main subsections. The first one sets the assumptions we have taken over our model in regard to the previous explained specification in Section 3 and the Dolev-Yao adversary model in Section 2.1. Second subsection talks about the symbolic model of Kyber build with Maude. Finally, third subsection shows how the model behaves using the online graphic tool Anima [37][38].

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 this assumptions in 2 categories: Dolev-Yao adversary assumptions and Kyber-specification assumptions.

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 in the network there will be only three participants, being two of them honest (Alice and Bob) and one adversary (Eve).

About Kyber almost all assumptions are over mathematical levels and low level concepts. This allows us to abstract 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 filfilling 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 further 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 centered binomial distribution whenever it is the case.

4.2 Module composition

The symbolic model of Kyber is composed by various modules, specifically 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

Following the previous list of functional modules we will explain the main contributions of each of them to our final symbolic model. It is important to clarify that the code listed in the following paragraphs are from the last version of our solution, which relates to Step 6 explained in Section 1.3.

DATA-TYPES: About this functional module we should remark the importance of all mathematical assumptions made into it. Specifically the three first assumptions that can be found in the previous section applies to this module. The module represents the low level components of our symbolic model towards the official specification. That is, here we represent matrices and vectors, and all its associated operations. For vectors operations for the concatenation, addition and subtraction of vectors are declared and defined. First, the concatenation of two vectors is a new vector, and is defined with symbol `||`. On the one hand, the sum of vectors is declared as a commutative and associative operation with the identity element 0. On the other hand, subtraction of two vectors is defined as associative. About the *dot* product we declared it as associative only.

About matrices, we declared and defined the product operation between a matrix and a vector, giving as a result a new vector. Product of matrices and vectors is associative and we the defined following property over the *dot* product with the Eq. 4. Symbols *V1* and *V2* are vectors, and *M2* is a matrix.

$$(M1\, m*\ V1)\, dot\, V2 \ =\ V1\, dot\, (M1\, m*\ V2) \tag{4}$$

The transpose function is defined both for vectors and matrices, and also the distributive properties of them over operations such as product or addition of vectors and matrices is formalized in the Eq. 5. Once again, symbols are the same as the previous equation, but now we specify two cases of distribution. The upper equation defines the distribution of the transpose vector over the sum of vectors. The lower defines the distribution of the transpose vector over the matrix-vector product.

$$\begin{aligned} tpV(V1\, v+\ V2) \ &=\ tpV(V1)\, v+\ tpV(V2) \\ tpV(M1\, m*\ V1) \ &=\ tpM(M1)\, m*\ tpV(V1) \end{aligned} \tag{5}$$

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 it is in this module where 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 constants values are non zero natural numbers that take a specific value depending on the version of Kyber they are used, four us, they are constant 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.

KYBER-HASH-OPERATIONS: On this functional module it is important to mention its necessity to obtain the previously declared constants, *Rho* and *Sigma*, through a new declared constant *d* using a new hash function *G* completely define for three possible scenarios of inputs and outputs. The module also contains a hash-function *H* but it is not defined and is only used for representation uses so the symbolic specification resembles the computational specification.

```
--- Property of noise cancellation
eq  (V1 v+ Decompress(X,N)) v- V2 = Decompress(X,N) .
```

Figure 13: Definition of the property to cancel the noise present in the component of a ciphered text *c* in functional module *ENCRYPTION*.

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 property showed in Subsection 3.2. Equations available in Eq. 6 represent the property of decompressing a compressed content and vice versa. The reason about the commutativity of the property is 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.

$$\begin{aligned} Decompress(Compress(X,N),N) = X \\ Compress(Decompress(X,N),N) = X \end{aligned} \tag{6}$$

We also specify the property of noise cancellation that represents the ideal case where both noises from *u'* and *v'* in the Eq. 7 are properly canceled because of subtraction. The following Fig. 13 shows how we modeled it so when we have a compressed value and two vectors they cancel each other as the noise equation specified.

$$\begin{aligned} Compress(v' - s^T u', 1) \\ Compress(Decompress(X,N),N) = X \end{aligned} \tag{7}$$

KYBER-CPAPKE-KEYGEN: The functional module samples the necessary values for the key generation function *KeyGen*. We declared the three needed sampling operations *generateA*, *sampleS* and *sampleE*. Each one of them has been defined in three possible inputs and outputs in order to model the values obtained depending on the provided value by the participant. It should be noted that function *generateA* is deterministic, which Maude can well represent thanks to equations, but functions *sampleS* and *sampleE* are in reality nondeterministic following a centered binomial distribution. This problem has been corrected with the assumptions made in Section 4.1.

KYBER-CPAPKE-ENC: This is the last functional module and it provides with the necessary operators to the sample values for step *Enc* in the Kyber scheme. Sampling functions behave the similar as in the previous functional module, with three different definitions for each one. The main difference lies in the sampled functions and the values represented with the defined constants. Now we have *smapleR'*, *sampleE1* and *sampleE2*. All of them receive a vector and output another vector representing a sampled value. Also, all of them are assumed to sample from a centered binomial distribution as we did with the previous sample functions.

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

```
ops Alice Eve Bob : ->  Identifier .

op _[_]_ : Identifier Keys Content -> Principal .
```

Figure 14: Definition of a participant at the network in our system module KYBERV2.

in our model for participants, messages and the global state of the system. Then the behaviour of the participants is explained, and finally pass over to the definition of intruder capabilities.

Element definitions Participants in our model follow the operational definition showed in Fig. 14. Here three identifiers are declared for our corresponding participants of the same name. Then the structure of a participant is declared. A participant consists of an *Identifier*, like the ones that has 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, id est, in its memory. About the key pairing we defined two kinds of keys: public keys and secret keys. The former ones are defined to be shared and known by any participant, even if it is malicious. The later are only known by the creator of the key. In regards to `content` types a participant could have ciphered texts or sampled values like *r* or *m* associated to its identifier.

All of these elements have concrete operators to add information of interest. For example, we have defined operators *publicKey(ID,PK) and secretKey(ID,SK)* to state in a group of keys who is the owner of the key, in other words, who generated it. Another primitive of interest is *sharedKey(ID,K)* to depict that the key *K* is shared with participant *ID*. We also have functions for knowing the author of a ciphered text with *cI(ID,C)*, or for the sample values, following the same structure as the one before but with the sampled value at the beginning before the *I*, id est *rI(ID,V)* for the *r* values.

Messages are defined by the operator showed in Fig. 15. A message contains information about two participant identifiers, a status of the message and the content it carries. First, the former identifier indicates the source of the message, and the later is the identifier of the participant to whom the messages is delivered throw the network. Then, status of a message can take the values showed on the second line of Fig. 15. A differentiation between states have been made in order to avoid unwanted behaviour of the network. 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. It is worth mentioning the definition of a function `in` receiving a message and a group of messages. Such function behaviour is to tell if a message is present in the message group. Such checking is of interest for the specification of participant behaviour while sending and receiving messages.

At last, the definition of the structure that represents our system can be seen in Fig. 16. Here, and from left to right, are specified the elements our rules will handle. At the most right corner we assigned a field for all the sample values available, that is constant values representing vector values in order 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 messages that is associative, representing in this way a kind of record that lets participants work over the sent messages.

Participant behaviour: Following the Kyber specification [4] and the Dolev-Yao assumptions we specify the following rules in Maude to model how the protocol operates. All these rules have been

```
--- Posible states for a message delivering PK
ops sentPK receivedPK interceptedPK : -> MsgState .

--- Posible states for a message delivering C
ops sentC receivedC interceptedC : -> MsgState .

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

Figure 15: Definition of a message at the network in our system module KYBERV2.

```
--- Global state composed by Samples, Principals and a Network
--- with messages
op {_}<_>net(_) : Content Principals Msgs -> GlobalState .
```

Figure 16: Definition of the global state of the system in our system module KYBERV2.

written in order to be as much general as possible, making the model and the constructed execution tree more realistic and interesting for model checking.

The first rule is *KeyGen* as can be seen in Fig. 17. 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. The construction of both keys, public and secret, is done by means of the matching equations in the conditions of the rule. The structure is the one present at the specification and can be seen in Fig. 12, 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 centered binomial distribution, so no further operations are needed for its computation.

Rule *SendPK* is showed in Fig. 18 and it models the behaviour of a participant with his own public key sending it to any other participant in the network different than him. The message is sent in case it hasn't been sent previously so infinite execution is avoided.

About rule *RecievePK*, its purpose is to process all incoming messages that contain a public key if it has not been received yet. Fig. 19 shows the code where we can see that only the last message on the

```
--- For now the protocol only starts when the participant can
--- sample necessary values for KeyGen step
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 17: Definition of conditional rule *KeyGen* in our system module KYBERV2.

```
crl [SendPK] : { CONT } < (ID1[publicKey(ID1,PK) ; KS1]dI(ID1,SAM1)
CONT1) (ID2[KS2]CONT2) (ID3[KS3]CONT3) >net(MSGS) =>
{ CONT } < (ID1[publicKey(ID1,PK) ; KS1]dI(ID1,SAM1)
CONT1) (ID2[KS2]CONT2) (ID3[KS3]CONT3) >net(MSGS msg{(ID1,ID2)
[sentPK]PK})
        if (msg{(ID1,ID2)[sentPK]PK}) in MSGS == false /\
        (msg{(ID1,ID2)[interceptedPK]PK}) in MSGS == false /\
        (msg{(ID1,ID2)[receivedPK]PK}) in MSGS == false /\
        (ID1 =/= ID2) /\
        (msg{(ID1,ID3)[sentPK]PK}) in MSGS == false /\
        (msg{(ID1,ID3)[interceptedPK]PK}) in MSGS == false /\
        (msg{(ID1,ID3)[receivedPK]PK}) in MSGS == false .
```

Figure 18: Definition of conditional rule *SendPK* in our system module KYBERV2.

```
crl [RecievePK] : { CONT } < (ID2[KS2]CONT2) PS >net(MSGS
msg{(ID1,ID2)[sentPK]PK}) => { CONT } < (ID2[publicKey(ID1,PK) ;
KS2]CONT2) PS >net(MSGS msg{(ID1,ID2)[receivedPK]PK})
        if (msg{(ID1,ID2)[receivedPK]PK}) in MSGS == false .
```

Figure 19: Definition of conditional rule *RecievePK* in our system module KYBERV2.

network is processed. This means that if a new message is sent before another is received, then it will be 'lost' to the participant. This has been done to simplify the protocol behaviour over the network.

Rule *Enc* as it is shown in Fig. 20 models the function with its same name. In order to apply the encryption step, a participant first has to know the public key of a participant, that means he must have received if before. 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. On application of the rule, the participant posses in the pool of keys a new shared key containing the value to be transmitted securely to the other participant. In the content pool, the ciphered text containing such 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 present in Fig. 12.

The counterpart of *SendPK* but for a ciphered text *c* obtained in *Enc* is the conditional rule *SendCiph* shown in Fig. 21. It checks similar conditions as when sending the public key so no infinite execution happens. It is important to mention that the difference between both rules is that in this one the message can only be sent with destiny the identification of the owner of the received public key.

The rule to receive a ciphered text, *RecieveCiph*, can be seen in Fig. 22. As with previous rule *SendCiph*, it behaves similarly to its own 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, rule to decipher the received cryptogram is *Dec*. In Fig. 23 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

```
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 20: Definition of conditional rule *Enc* in our system module KYBERV2.

```
crl [SendCiph] : { CONT } < (ID2[sharedKey(ID1,SAM1) ; publicKey(ID1,
(M1 m* V1) v+ V2) ; KS1]cI(ID2,C) rI(ID2,SAM2) CONT1) PS >net(MSGS) =>
{ CONT } < (ID2[sharedKey(ID1,SAM1) ; publicKey(ID1,
(M1 m* V1) v+ V2) ; KS1]cI(ID2,C) rI(ID2,SAM2) CONT1) PS >
net(MSGS msg{(ID2,ID1)[sentC](C)})
        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) /\
        (msg{(ID2,ID1)[sentC](C)}) in MSGS == false /\
        (msg{(ID2,ID1)[interceptedC](C)}) in MSGS == false /\
        (msg{(ID2,ID1)[receivedC](C)}) in MSGS == false .
```

Figure 21: Definition of conditional rule *SendCiph* in our system module KYBERV2.

```
crl [RecieveCiph] : { CONT } < (ID1[KS1]CONT1) PS >net(MSGS
msg{(ID2,ID1)[sentC](C)}) => { CONT } < (ID1[KS1]cI(ID2,C) CONT1)
PS >net(MSGS msg{(ID2,ID1)[receivedC](C)})
        if (msg{(ID2,ID1)[receivedC](C)}) in MSGS == false .
```

Figure 22: Definition of conditional rule *RecieveCiph* 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 23: Definition of conditional rule *Dec* in our system module KYBERV2.

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

Figure 24: Definition of a rule *Comp* in our system module KYBERV2.

is applied, obtaining the shared key *K1*.

There is one last rule, which is not conditional, that represents the execution's ending of the system once two participants have securely shared a key between them. This rule can be seen in Fig. 24 and its name is *Comp*. This rule checks if a participant with identifier *ID1* has in his group of keys a shared key *K1* whit a participant *ID2*, and the other participant with identifier *ID2* happens to have the same shared key *K1* with *ID1*. When the global state matches with this premise, it turns to the constant operator `True` indicating the protocol execution has succeeded.

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

The former one can be seen in Fig. 25, 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, and the intruder places a new message identical to the previous one but with his own public key as content. This will make the receiver think the public key received is from the sender when it is not, thus beginning the man in the middle attack.

The later is available in Fig. 26, and makes the intruder intercept a message sent with a ciphered text. This intercepted message is the one 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

```
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 25: 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 26: Definition of conditional rule *Intercept2* in our system module KYBERV2.

ciphered texts, his own and the one intercepted.

4.3 Anima

Now we will show how the model runs interactively using the online web tool Anima [37] [38]. We only show the interaction between two participants sharing a key, starting with an initial state of three participants and sufficient sampling values for one key exchange, in other words, one execution of the protocol.

The execution trace starts with the initial state at the top of the tree, as Fig. 27 shows. From there, three main possibilities appear, each of them consisting of applying the conditional rule *KeyGen* but on different participants. We chose in this case *Alice* to be the one to generate the pair of public and secret keys. Then, two new possibilities arise. It is the rule *SendPK* to represent Alice sending his public key either to *Bob* or *Eve*, the other participants present in the network. We chose the path of *Alice* sending *PK* to *Bob* because of standard reasons, but following assumptions taken it is possible that a honest party wants to arrange a secret key with an intruder, even if he doesn't know their malicious intentions. From that point onwards the execution is almost linear, performing the *Receiving* and *Enc* rules.

Then, once the ciphered text has been generated, rules to send and receive it are applied. Note that in Fig. 28 only one possible path is available. That is because how we modeled the protocol so the ciphered text is only sent to whom sent the first message containing the public key.

Finally, the end of the execution is shown in Fig. 29, where the *Dec* step rule takes place to make both, *Alice* and *Bob*, share the same key *m*. The `True` state indicates that a shared key has been **securely** established between two participants.

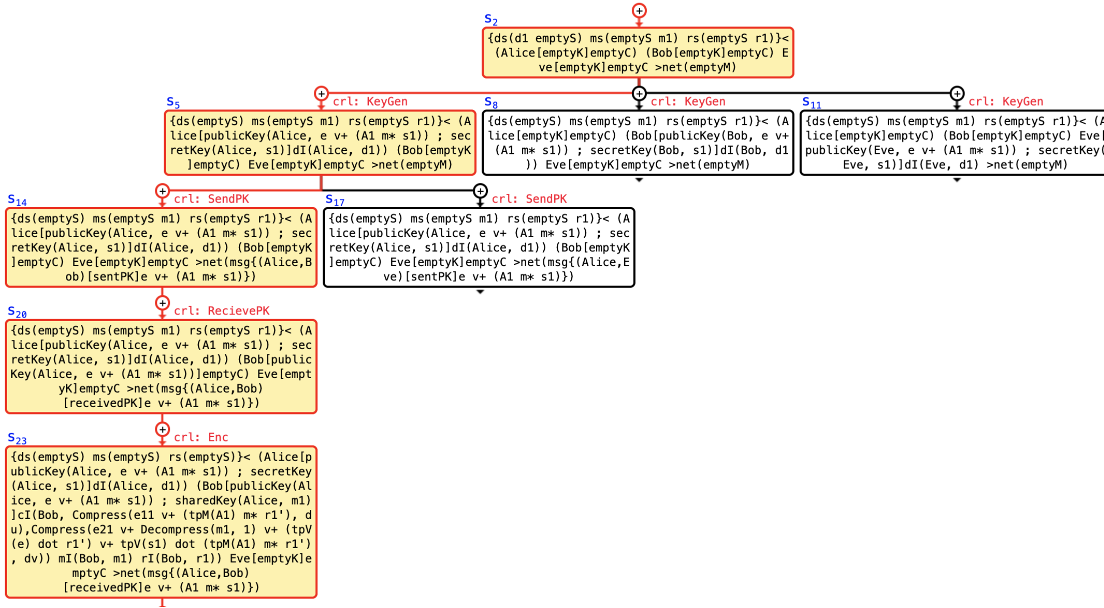

Figure 27: Execution trace of our symbolic model from *init* state till step *Enc* showed by the online web tool Anima.

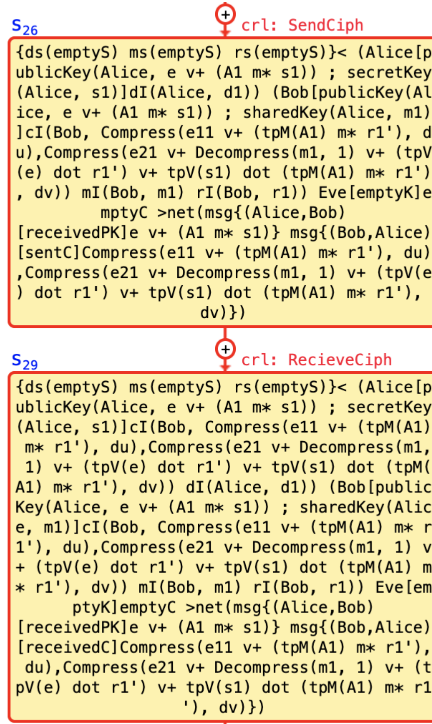

Figure 28: Execution trace of our symbolic model from *SendCiph* till step *ReceieveCiph* showed by the online web tool Anima.

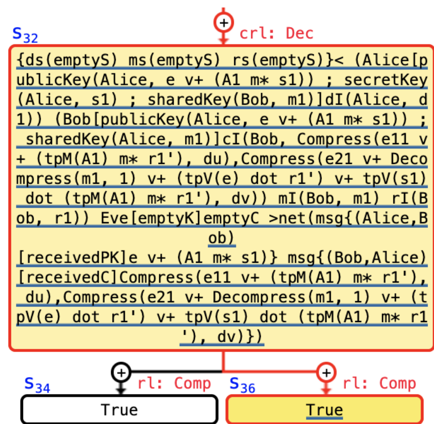

Figure 29: Execution trace of our symbolic model on *Dec* step showed by the online web tool Anima.

.

Chapter 5

5 Maude verification

In this chapter we take a look at the two approximations taken to perform verification of our symbolic model. First we see a more informal verification using the *search* command from Maude to search from two different initial states to check if a key is shared between two participant securely, and also check if a man-in-the-middle attack is present. Second we dive in to a more formal form of verification using the Model Checking capabilities of Maude. We define some properties and prove the correctness of our model with some LTL formulas.

5.1 Reachability verification

What we are going to verify using the *search* command is if the model behaves as expected, that means, check if there exist states of interest. For that goal we conduct reachability analysis from two initial states, *init* and *init2*, both available in Fig. 30. The initial state *init* defines our global state with three types of samples, each one with a value available for sampling. It also specifies that the network is populated by three participants 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*.

```
ops init init2 : -> GlobalState .

eq init = {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 30: Declaration and definition of two initial states for our system module KYBERV2 in Maude.

For each of these initial states we check 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 achieved once in the global state there are two participants with the same shared key between them.
- If there exists a state in the state space tree in which a man in the middle attack has happened. It is achieved with the command in Fig. 31, where 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.

```
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 31: Command template for a man-in-the-middle attack search in Maude.

```
==========================================
search in KYBERV2 : init =>* True .

Solution 1 (state 40)
states: 41  rewrites: 1225 in 1ms cpu (2ms real) (626598 rewrites/second)
empty substitution

No more solutions.
states: 41  rewrites: 1396 in 2ms cpu (2ms real) (624050 rewrites/second)
==========================================
search in KYBERV2 : init =>* {CONT}< (ID1[KS1 ; sharedKey(ID3, K1)]CONT1) (ID2[
    KS2 ; sharedKey(ID1, K1) ; sharedKey(ID3, K2)]CONT2) ID3[KS3 ; sharedKey(
    ID1, K2)]CONT3 >net(MSGS) .

No solution.
states: 41  rewrites: 1396 in 1ms cpu (2ms real) (703274 rewrites/second)
```

Figure 32: Results on the reachability analysis over the first initial state in module KYBERV2.

On the first initial state we can see in Fig. 32 the result of applying two *search* command. The results are that for *init* there is a state where two participant have shared a key, meaning the protocol works. And 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.

About the second initial state, we can see in Fig. 33 the results of both search commands applied over it. As well as in the first state, state two also 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 different possibilities the model gives 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 wen *Alice* or *Bob* try to use those secret keys to communicate to each other they will be in deed 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

In this subsection we conduct a more formal verification of our symbolic model with the model checker over LTL formulas. We first explain the predicates specified in the system module *KYBERV2-PREDS*. Then we show the defined LTL formulas using those predicates to define some properties over our system module. Finally, we show the use of model checking and reason the obtained results.

```
search in KYBERV2 : init2 =>* True .

Solution 1 (state 4049)
states: 4050  rewrites: 157896 in 174ms cpu (176ms real) (902422
    rewrites/second)
empty substitution

No more solutions.
states: 68126  rewrites: 8412735 in 10884ms cpu (11037ms real) (772888
    rewrites/second)
==========================================
search in KYBERV2 : init2 =>* {CONT}< (ID1[KS1 ; sharedKey(ID3, K1)]CONT1) (
    ID2[KS2 ; sharedKey(ID1, K1) ; sharedKey(ID3, K2)]CONT2) ID3[KS3 ;
    sharedKey(ID1, K2)]CONT3 >net(MSGS) .

Solution 1 (state 34000)
states: 34001  rewrites: 2394038 in 2997ms cpu (3027ms real) (798636
    rewrites/second)
CONT --> ds(emptyS) ms(emptyS) rs(emptyS)
ID1 --> Alice
KS1 --> publicKey(Alice, e v+ (A1 m* s1)) ; secretKey(Alice, s1)
ID3 --> Bob
K1 --> m1
CONT1 --> dI(Alice, d1)
ID2 --> Eve
KS2 --> publicKey(Alice, e v+ (A1 m* s1)) ; publicKey(Eve, e' v+ (A2 m* s2)) ;
    secretKey(Eve, s2)
K2 --> m2
CONT2 --> dI(Eve, d2) mI(Eve, m1) rI(Eve, r1) cI(Eve, Compress(e11 v+ (tpM(A1)
    m* r1'), du),Compress(e21 v+ (tpV(s1) dot (tpM(A1) m* r1')) v+ (tpV(e) dot
    r1') v+ Decompress(m1, 1), dv))
KS3 --> publicKey(Alice, e' v+ (A2 m* s2))
CONT3 --> mI(Bob, m2) rI(Bob, r2) cI(Bob, Compress(e12 v+ (tpM(A2) m* r2'),
    du),Compress(e22 v+ (tpV(s2) dot (tpM(A2) m* r2')) v+ (tpV(e') dot r2') v+
    Decompress(m2, 1), dv))
MSGS --> msg{(Alice,Bob)[interceptedPK]e v+ (A1 m* s1)} msg{(Alice,Bob)[
    receivedPK]e' v+ (A2 m* s2)} msg{(Bob,Alice)[interceptedC]Compress(e12 v+ (
    tpM(A2) m* r2'), du),Compress(e22 v+ (tpV(s2) dot (tpM(A2) m* r2')) v+ (
    tpV(e') dot r2') v+ Decompress(m2, 1), dv)} msg{(Bob,Alice)[
    receivedC]Compress(e11 v+ (tpM(A1) m* r1'), du),Compress(e21 v+ (tpV(s1)
    dot (tpM(A1) m* r1')) v+ (tpV(e) dot r1') v+ Decompress(m1, 1), dv)}
```

Figure 33: Results on the reachability analysis over the second initial state in module KYBERV2.

```
*** Declaration of predicate to represent a principal starting the
*** protocol
op wantsToShareKey : Identifier Identifier -> Prop .

*** Declaration of predicate to represent a principal finished the
*** protocol
op sharedAKeyWith : Identifier Identifier -> Prop .

*** Declaration of predicate to represent a principal stole a
*** secret key
op stolenSecret : Identifier Identifier -> Prop .
```

Figure 34: Declaration of three predicates in KYBERV2-PREDS.

```
*** Equations indicating two actors want to share a key
eq { CONT } < (ID1[publicKey(ID1,PK) ; KS1]CONT1) (ID2[KS2]CONT2)
PS >net(MSGS msg{(ID1,ID2)[sentPK]PK}) |=
wantsToShareKey(ID1,ID2) = true .
```

Figure 35: Definition of predicate *wantsToShareKey* in KYBERV2-PREDS.

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 properties about our system. The system module has already been defined and tested, and in this subsection we dive into the modules needed to do model checking, specially for predicate definition. The predicates we have specified are three, all of them in the system module *KYBERV2-PREDS*. In Fig. 34 we have declared three predicates. The three of them receive two identifiers in order to select the participants involved in the predicate.

Predicate *wantsToShareKey* is defined, see Fig. 35, 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 represents that a participant wants to share a key with another one, in other words, it is the start of the protocol Kyber.

Predicate *sharedAKeyWith* is defined in Fig. 36 so it is true when there are two participants and each of them holds the same shared key for the other one. This predicate represents the end of the protocol, fullfiling the previous predicate in which the protocol was started.

Finally, last predicate *stolenSecret* is defined in Fig. 37, and states that a secret key has been stolen

```
*** Equations indicating two actors share a key
eq { CONT } < (ID1[sharedKey(ID2,K1) ; KS1]CONT1)
(ID2[sharedKey(ID1,K1) ; KS2]CONT2) PS >net(MSGS) |=
sharedAKeyWith(ID1,ID2) = true .
```

Figure 36: Definition of predicate *sharedAKeyWith* in KYBERV2-PREDS.

```
*** Equations indicating an actor learned the secret key from
*** another one
eq { CONT } < (ID1[secretKey(ID1,SK) ; KS1]CONT1)
(ID2[secretKey(ID1,SK) ; KS2]CONT2) PS >net(MSGS) |=
stolenSecret(ID1,ID2) = true .
```

Figure 37: Definition of predicate *stolenSecret* in KYBERV2-PREDS.

```
[]<> wantsToShareKey(Alice,Bob)) -> ([]<> sharedAKeyWith(Alice,Bob))
```

Figure 38: Definition of a property about liveness for our symbolic model KYBERV2.

from a participant if it is also contained in the pool of keys of a different participant.

5.2.2 Properties

With the predicates defined we specify now two properties. One checks the liveness of our protocol and the other has to do with it's security. These properties are LTL formulas that allow us to explore the execution tree in search for counterexamples that do nor satisfies the formulas, and thus proving the property does not hold. If no counter example is found we can say with assurance that the symbolic model satisfies the given property.

The first property, shown in Fig. 38, 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, shown in Fig. 39, has to do with the assurance that in any future state the predicate of *stolenSecret* is true. This means that 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: *initial1* and *initial2*. The results can be seen in both Fig. 40 and Fig. 41. On the former figure we can see that both initial states accomplish liveness property when *Alice* and *Bob* want to share a key. On the later figure, 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.

```
[] ~(stolenSecret(Alice,Eve))
```

Figure 39: Definition of a property for security for our symbolic model KYBERV2.

```
Maude> red modelCheck(initial1, ([]<> wantsToShareKey(Alice,Bob)) -> ([]<> sharedAKeyWith(Alice,Bob))) .
reduce in KYBERV2-CHECK : modelCheck(initial1, []<> wantsToShareKey(Alice, Bob) -> []<> sharedAKeyWith(Alice, Bob)) .
rewrites: 1469 in 0ms cpu (2ms real) (1883333 rewrites/second)
result Bool: true
Maude> red modelCheck(initial2, ([]<> wantsToShareKey(Alice,Bob)) -> ([]<> sharedAKeyWith(Alice,Bob))) .
reduce in KYBERV2-CHECK : modelCheck(initial2, []<> wantsToShareKey(Alice, Bob) -> []<> sharedAKeyWith(Alice, Bob)) .
rewrites: 8425427 in 14554ms cpu (14577ms real) (578884 rewrites/second)
result Bool: true
```

Figure 40: Results on the LTL satisfability over the liveness property over the symbolic model for Kyber.

```
Maude> red modelCheck(initial1, ([] ~(stolenSecret(Alice,Eve)))) .
reduce in KYBERV2-CHECK : modelCheck(initial1, []~ stolenSecret(Alice, Eve)) .
rewrites: 1404 in 0ms cpu (2ms real) (8886075 rewrites/second)
result Bool: true
Maude> red modelCheck(initial2, ([] ~(stolenSecret(Alice,Eve)))) .
reduce in KYBERV2-CHECK : modelCheck(initial2, []~ stolenSecret(Alice, Eve)) .
rewrites: 8412743 in 10757ms cpu (10820ms real) (782015 rewrites/second)
result Bool: true
```

Figure 41: Results on the LTL satisfability over the security property over the symbolic model for Kyber.

.

Chapter 6

6 Conclusion and future work

In this master's thesis we have proven the presence of a man in the middle attack on the key encapsulation mechanism Kyber given Dolev-Yao adversary assumptions. Moreover, some LTL formulas, specifying both liveness and security properties have been applied with Maude's LTL Logical Model Checker in order to extend the verification of our model.

In order to achieve this we had to first define a formal specification in the form of a symbolic model in Maude using its functional and system modules in a modular manner, so it is more clear and intuitive. Later we performed reachability analysis with the `search` command, where the model was proven to function and a the man-in-the-middle was found. Later we defined extra modules to define predicates which then where used to define properties with LTL formulas. Such properties were applied for given actors of the protocol and verified that two honest actors can end sharing a key as the specification in [4] states, and that the secret key can not be stolen by the adversary.

Our results prove that the key encapsulation mechanism Kyber is not safe from classical adversaries if no form of authentication is available, thus breaking the scheme. Finally, with this new symbolic model we have provided a new approximation to represent the system different from [21]. We also have provided and applied a new analysis approach using the logical model checker, thus extending the reachability analysis made by [21].

As future work, we plan now to improve the model by conducting extended model checking in order to verify its complete correctness. We also want to extend the model to better represent the key encapsulation mechanism. 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 master's thesis, 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 also have in mind the use of protocol analysis tools, such as MaudeNPA, to specify the protocol and check its security in a more thoughtful analysis form of the system. This would 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, and R. Steinwandt, "Applying grover's algorithm to aes: quantum resource estimates," in *Post-Quantum Cryptography*. Springer, 2016, pp. 29–43.

[3] T. Chou, C. Cid, S. UiB, J. Gilcher, T. Lange, V. Maram, R. Misoczki, R. Niederhagen, K. Paterson, and E. Persichetti, "Classic mceliece: conservative code-based cryptography, 10 october 2020," 2020.

[4] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber algorithm specifications and supporting documentation," *NIST PQC Round*, vol. 2, no. 4, 2019.

[5] C. Chen, O. Danba, J. Hoffstein, A. Hülsing, J. Rijneveld, J. M. Schanck, P. Schwabe, W. Whyte, and Z. Zhang, "Algorithm specifications and supporting documentation," *Brown University and Onboard security company, Wilmington USA*, 2019.

[6] J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, and F. Vercauteren, "Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem," in *International Conference on Cryptology in Africa*. Springer, 2018, pp. 282–305.

[7] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of crystals-dilithium," in *2021 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2021, pp. 1–10.

[8] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over ntru," *Submission to the NIST's post-quantum cryptography standardization process*, vol. 36, no. 5, 2018.

[9] J. Ding and D. Schmidt, "Rainbow, a new multivariable polynomial signature scheme," in *International conference on applied cryptography and network security*. Springer, 2005, pp. 164–175.

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

[11] V. Cortier, S. Kremer, and B. Warinschi, "A survey of symbolic methods in computational analysis of cryptographic systems," *Journal of Automated Reasoning*, vol. 46, no. 3, pp. 225–259, 2011.

[12] M. Abadi and P. Rogaway, "Reconciling two views of cryptography (the computational soundness of formal encryption)," *Journal of cryptology*, vol. 15, no. 2, pp. 103–127, 2002.

[13] S.-L. Gazdag, S. Grundner-Culemann, T. Guggemos, T. Heider, and D. Loebenberger, "A formal analysis of ikev2's post-quantum extension," in *Annual Computer Security Applications Conference*, 2021, pp. 91–105.

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

[15] S. Escobar, C. Meadows, and J. Meseguer, "Maude-npa: Cryptographic protocol analysis modulo equational properties," in *Foundations of Security Analysis and Design V*. Springer, 2009, pp. 1–50.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Springer, 2007, vol. 4350.

[17] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, "Proverif 2.00: automatic cryptographic protocol verifier, user manual and tutorial," *Version from*, pp. 05–16, 2018.

[18] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *International conference on computer aided verification*. Springer, 2013, pp. 696–701.

[19] 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.

[20] J. Ramsdell and J. Guttman, "CPSA4: A cryptographic protocol shapes analyzer," https://github.com/mitre/cpsaexp, The MITRE Corporation, 2018.

[21] D. D. Tran, K. Ogata, S. Escobar, S. Akleylek, and 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.

[22] ——, "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, to appear.

[23] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, pp. 198–208, 1983.

[24] M.-O. Stehr, J. Meseguer, and P. C. Ölveczky, "Rewriting logic as a unifying framework for petri nets," in *Unifying Petri Nets*. Springer, 2001, pp. 250–303.

[25] N. M. Oliet and J. A. V. 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.

[26] J. Meseguer, "A logical theory of concurrent objects and its realization in the Maude language," in *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner, and A. Yonezawa, Eds. MIT Press, 1993, pp. 314–390.

[27] M. Katelman, S. Keller, and J. Meseguer, "Rewriting semantics of production rule sets," *Journal of Logic and Algebraic Programming*, vol. 81, no. 7-8, pp. 929–956, 2012.

[28] S. Liu, P. C. Ölveczky, and J. Meseguer, "Modeling and analyzing mobile ad hoc networks in Real-Time Maude," *Journal of Logical and Algebraic Methods in Programming*, 2015.

[29] R. Bobba, J. Grov, I. Gupta, S. Liu, J. Meseguer, P. Ölveczky, and S. Skeirik, "Design, formal modeling, and validation of cloud storage systems using Maude," in *Assured Cloud Computing*, R. H. Campbell, C. A. Kamhoua, and K. A. Kwiat, Eds. J. Wiley, 2018, ch. 2, pp. 10–48.

[30] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and 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.

[31] J. Meseguer and G. Roșu, "The rewriting logic semantics project," *Theoretical Computer Science*, vol. 373, pp. 213–237, 2007.

[32] K. Bae, J. Meseguer, and P. C. Ölveczky, "Formal patterns for multirate distributed real-time systems," *Science of Computer Programming*, vol. 91, pp. 3–44, 2014.

[33] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and K. Sonmez, "Pathway logic: Symbolic analysis of biological signaling," in *Biocomputing 2002*. World Scientific, 2001, pp. 400–412.

[34] C. Talcott, S. Eker, M. Knapp, P. Lincoln, and K. Laderoute, "Pathway logic modeling of protein functional domains in signal transduction," in *Biocomputing 2004*. World Scientific, 2003, pp. 568–580.

[35] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, vol. 96, no. 1, pp. 73–155, 1992.

[36] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The maude ltl model checker," *Electronic Notes in Theoretical Computer Science*, vol. 71, pp. 162–187, 2004.

[37] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña, "Exploring conditional rewriting logic computations," *Journal of Symbolic Computation*, vol. 69, pp. 3–39, 2015.

[38] "Anima online stepper," http://safe-tools.dsic.upv.es/anima/, accessed: 08-07-2022.