Hindawi
IET Information Security
Volume 2023, Article ID 9399887, 17 pages
https://doi.org/10.1049/2023/9399887

Research Article

Kyber, Saber, and SK-MLWR Lattice-Based Key Encapsulation Mechanisms Model Checking with Maude

Duong Dinh Tran,[1] **Kazuhiro Ogata**,[1] **Santiago Escobar**,[2] **Sedat Akleylek**,[3,4] **and Ayoub Otmani**[5]

[1]*Japan Advanced Institute of Science and Technology, Ishikawa 923-1292, Japan*
[2]*VRAIN, Universitat Politècnica de València, Valencia, Spain*
[3]*Ondokuz Mayis University, Samsun, Türkiye*
[4]*University of Tartu, Tartu, Estonia*
[5]*University of Rouen Normandie, Rouen, France*

Correspondence should be addressed to Duong Dinh Tran; duongtd@jaist.ac.jp

Received 13 June 2023; Revised 8 September 2023; Accepted 11 September 2023; Published 30 October 2023

Academic Editor: Thomas Haines

Facing the potential threat raised by quantum computing, a great deal of research from many groups and industrial giants has gone into building public-key post-quantum cryptographic primitives that are resistant to the quantum attackers. Among them, there is a large number of post-quantum key encapsulation mechanisms (KEMs), whose purpose is to provide a secure key exchange, which is a very crucial component in public-key cryptography. This paper presents a formal security analysis of three lattice-based KEMs including Kyber, Saber, and SK-MLWR. We use Maude, a specification language supporting equational and rewriting logic and a high-performance tool equipped with many advanced features, such as a reachability analyzer that can be used as a model checker for invariant properties, to model the three KEMs as state machines. Because they all belong to the class of lattice-based KEMs, they share many common parts in their designs, such as polynomials, vectors, and message exchange patterns. We first model these common parts and combine them into a specification, called base specification. After that, for each of the three KEMs, by extending the base specification, we just need to model some additional parts and the mechanism execution. Once completing the three specifications, we conduct invariant model checkings with the Maude search command, pointing out a similar man-in-the-middle attack. The occurrence of this attack is due to the fact that authentication is not part of the KEMs, and therefore an active attacker can modify all communication between two honest parties.

1. Introduction

Quantum attack threat, of which Shor's [1] algorithm is known as the most effective one, is a credible threat affecting most public-key (or asymmetric) cryptosystems in use today. The essential reason is that the security of those systems relies on some specific hard mathematical problems (e.g., the integer factorization problem) which are intractable for conventional computers to solve, but can be efficiently solved by a sufficiently large quantum computer running Shor's algorithm. Because of the steady development of quantum computers in recent years with the participation of many giants, such as IBM, Google, and Microsoft, the quantum attack threat to the public-key cryptosystems may happen in near future. On the other hand, quantum computers pose less danger to the symmetric primitives. Although the complexity of breaking symmetric primitives can be made simpler using Grover's algorithm [2], these attacks can be effectively avoided by doubling the key size. In particular, AES-256 would be as hard to break by a quantum computer as AES-128 is by a classical computer.

Facing that potential quantum attack threat, a great deal of research has gone into constructing new cryptosystems that are secure even in the presence of quantum attackers, so-called post-quantum cryptosystems. In 2017, the Post-Quantum

Cryptography Project was launched by the National Institute of Standards and Technology (NIST), calling for submissions for post-quantum cryptographic algorithms (https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantumcryptography-standardization/Call-for-Proposals). There were 69 submissions in the first round of this standardization project, among them, many were post-quantum key encapsulation mechanism (KEM) proposals, which were dedicated to securely establishing a shared key between two parties over an insecure network. This is understandable because the symmetric key establishment is considered one of the most crucial components in the public-key cryptography.

Cryptographic protocol analysis can be largely classified into two complementary approaches [3]: computational analysis and symbolic analysis. The former treats messages as bit strings and cryptographic primitives as functions from bit strings to bit strings. A security proof in the computational approach can be regarded as a mathematical reduction, where the only chance to violate the security of the given cryptographic construction is to solve some presumed computationally infeasible problems. The latter treats messages as terms and cryptographic primitives as functions from terms to terms. The attacker's capabilities are modeled by manipulating terms representing messages exchanged in the network. Three KEMs considered in this paper were already given security proofs in the computational approach. Those computational proofs provide a tight security guarantee because it takes probability and complexity into account. However, they are not computer-verified, complicated in general, and difficult for those who are not cryptography experts to understand. Symbolic analysis, on the other hand, is easier to understand even for nonexperts, and more importantly, it is computer-verified and easily automated. The cost for that benefit is that perfect cryptography assumption is typically made in symbolic analysis. Our analysis reported in this paper belongs to the symbolic approach. Note that the analysis can be applied to not only the three KEMs but also other KEMs and other kinds of primitives as well.

In this paper, we formally specify and model check three lattice-based KEMs including Kyber [4] (precisely CRYSTALS-Kyber), Saber [5], and SK-MLWR [6] (the KEM by Akeylek and Seyhan [6] is called SK-MLWR in the present paper). Kyber bases its security on the presumed hardness of solving the learning with error (LWE) problem, while the security of Saber and SK-MLWR relies on the presumed difficulty of the module learning with rounding (MLWR) problem. LWE provides security by adding "noise" to the inner product of a secret vector with a random public vector. When used for key exchange, the inverse operation tries to guess the secret vector from the received key. LWR is a variant of LWE, where one replaces random noise with deterministic rounding in order to increase efficiency. Kyber was selected as a candidate to be standardized for public-key encryption and key exchange in July 2022 by NIST (https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022). It is the unique candidate for public-key encryption and key exchange. We use Maude [7], a specification language supporting both equational and rewriting logic, to model the three KEMs as state machines. We first model plenty of common parts in the designs of the three KEMs, such as polynomials and vectors. Then, the complete formal specification of each KEM is achieved by modeling some additional parts that are specific to only that mechanism, followed by modeling its execution. With the attacker model, we follow the Dolev and Yao [8] model to specify intruder capabilities. Once the three formal specifications are completed, we conduct invariant model checkings in Maude by using the `search` command, finding a man-in-the-middle (MITM) attack for each of the three KEMs. Although this kind of attack is not a novel attack for KEMs, what we present in this paper illustrates one possible way to symbolically analyze KEMs. To gain a confident security assurance, the three KEMs must be deeply analyzed due to their relative newness. Our ultimate goal is to conduct security analysis/verification of post-quantum cryptographic protocols. Such protocols use post-quantum cryptographic primitives, such as the three KEMs considered in this paper. Thus, for protocol security analysis, formally specifying such primitives are necessary. What is reported in this paper is our initial step toward the goal.

This is an extended version of our previous paper by Tran et al. [9], where Kyber is the only one mainly described. Another paper of ours by Tran et al. [10] has also reported a similar experiment with the Saber mechanism. Our work presented in this paper benefits from these two previous studies. We reuse those two independent formal specifications to first construct a so-called base specification, where common parts in the designs of the three KEMs are specified. Then, the complete specification for each KEM is quickly finalized by extending the base specification. In this way, we can specify the three lattice-based KEMs with less effort.

1.1. Roadmap. The structure of the remaining of this paper is as follows: Section 2 gives some preliminaries, such as the definitions of KEM and state machine. Section 3 briefly explains the three lattice-based KEMs. Section 4 presents the base specification used for the three KEMs. Then, Section 5 presents how to complete the Maude specification of Kyber from the base specification and reports its model checking experiment. Section 6 briefly describes the complete specifications of Saber and SK-MLWR. Section 7 notes some remarks on the experiments we have conducted. Section 8 discusses some related work and finally, Section 9 summarizes the paper.

All of the Maude specifications and checking commands reported in this paper can be downloaded from the webpage (https://github.com/duongtd23/lattice-based-kems-mc).

2. Preliminaries

2.1. Key Encapsulation Mechanism (KEM). We start with the definition of a general KEM.

Definition 1. A key encapsulation mechanism consists of the following three algorithms:

(i) $\texttt{KeyGen}() \longrightarrow (pk, sk)$: a probabilistic *key generation* algorithm that outputs a public/secret key pair pk and sk.
(ii) $\texttt{Encaps}(pk) \longrightarrow (c, k)$: a probabilistic *encapsulation* algorithm that takes a public key pk as input and returns a ciphertext (or encapsulation) c and a key k.
(iii) $\texttt{Decaps}(c, sk) \longrightarrow k$: a (typically deterministic) *decapsulation* algorithm that takes a ciphertext c and a secret key sk as inputs and returns a key k.

We say a KEM is ϵ-*correct* if for all $(pk, sk) \leftarrow \texttt{KeyGen}()$ and $(c, k) \leftarrow \texttt{Encaps}(pk)$, it holds that:

$$\Pr[\texttt{Decaps}(c, sk) \neq k] \leq \epsilon$$

In this work, all KEMs are assumed to be 0-*correct*, meaning that Encaps and Decaps always correctly produce the same shared key k. In order to do security verification in the symbolic model, idealizing presumptions like that are often required. Because of the really small probability of Decaps-failure for each KEM, typically almost 0, we are not overidealizing when omitting such Decaps-failure cases. For instance, that failure probability of Kyber is below 2^{-140} [4], namely Kyber is ϵ-*correct* with $\epsilon < 2^{-140}$.

In the following, we briefly describe some mathematical notations used in Kyber, Saber, and SK-MLWR. The three KEMs share some common mathematical backgrounds because they all belong to the class of lattice-based KEMs. For instance, they are all relied on the polynomial rings, vectors, and matrices.

Let $\mathcal{B}$ denote the set $\{0, \ldots, 255\}$, i.e., the set of 8-bit unsigned integers (bytes). $\mathcal{B}^k$ and $\mathcal{B}^*$ denote the set of byte arrays of length k and the set of byte arrays of arbitrary length, respectively. Given two byte arrays a and b, $(a \| b)$ denotes the concatenation of a and b.

Let $\mathbb{Z}_q$ denote the ring of integers modulo q. R and R_q denote the polynomial ring $\mathbb{Z}[X]/(X^n + 1)$ and the quotient polynomial ring $\mathbb{Z}_q[X]/(X^n + 1)$, respectively. Given $x \in \mathbb{Q}$, where $\mathbb{Q}$ denotes the rational numbers set, $\lfloor x \rceil$ denotes rounding of x to the closest integer. R_q^k and $R_q^{k \times l}$ denote the set of vectors and matrices from R_q with dimensions of k and $k \times l$, respectively. Regular font letters denote elements in R or R_q (which includes $\mathbb{Z}$ and $\mathbb{Z}_q$), while bold lowercase letters denote vectors with coefficients in R or R_q. By default, all vectors are column vectors. Bold uppercase letters are matrices. Given a vector $\mathbf{v}$ (or matrix $\mathbf{A}$), it transpose is denoted by $\mathbf{v}^T$ (or $\mathbf{A}^T$). $\mathbf{v}[i]$ is i-th entry of the vector (with indexing starting at zero) and similarly, $\mathbf{A}[i][j]$ is the entry of row i and column j of the matrix.

2.2. State Machine and Maude. We turn to describe in a nutshell how to specify a state machine in Maude and use the Maude **search** command to perform invariant model checking.

Definition 2. A state machine $\mathcal{M}$ is a tuple of $\langle \mathcal{S}, \mathcal{I}, \mathcal{T} \rangle$, where $\mathcal{S}$ is a set of states, $\mathcal{I} \subseteq \mathcal{S}$ is a set of initial states, and $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ is a binary relation over states.

Definition 3. The set $\mathcal{R}$ of *reachable states* with respect to $\mathcal{M}$ is defined as follows:

(i) for each $s \in \mathcal{I}$, $s \in \mathcal{R}$
(ii) for each $(s, s') \in T$, if $s \in \mathcal{R}$, then $s' \in \mathcal{R}$

In this paper, to express a state of $\mathcal{S}$, we use a braced associative-commutative collection (AC-collection) of name-value pairs. A name-value pair is called an observable component. The juxtaposition operator is used as the constructor of AC-collection. For instance, $oc1$, $oc2$, $oc3$ is the AC-collection of three observable components $oc1$, $oc2$, and $oc3$. A state is of the form $\{oc1, oc2, oc3\}$.

In this paper, state transitions are specified as rewrite rules in Maude [7], a specification language supporting both equational and rewriting logic. Moreover, Maude is also a high-performance tool equipped with several formal analysis functionalities, such as a reachability analyzer and an LTL model checker. A conditional rewrite rule in Maude is in the following form:

```
crl [label] : a => b if ... /\ c_i /\ ... .
```

where *label* is the label of the rule, a and b are patterns of the source state and the successor state, respectively. c_i is part of the condition, which may be an equation $lc_i = rc_i$. If the condition ... /\ c_i /\ ... holds under some substitution σ, $\sigma(a)$ can be replaced with $\sigma(b)$.

For the reachability analysis, Maude provides the **search** command that can find states reachable from a given state, matching a given pattern, and satisfying a given condition. A **search** command is in the following form:

```
search [n,m] in M : i =>* p such that c.
```

where M is the name of the Maude module specifying the state machine. n and m are optional arguments denoting a bound on the number of desired solutions and the maximum depth of the search, respectively. i, p, and c denote the given state, the pattern, and the condition(s), respectively. In practice, n typically is 1 and i typically represents an initial state of the state machine.

3. Kyber, Saber, and SK-MLWR

Through this section, we briefly describe the three KEMs.

3.1. Kyber. Figure 1 depicts the triple algorithms (KeyGen, Encaps, and Decaps) of Kyber KEM [4]. These algorithms employ three other algorithms (KeyGen, Enc, and Dec) of Kyber.PKE (stands for public key encryption), which are depicted in Figure 2. There are two hash functions $\mathcal{H} : \mathcal{B}^* \longrightarrow \mathcal{B}^{32}$ and $\mathcal{G} : \mathcal{B}^* \longrightarrow \mathcal{B}^{32} \times \mathcal{B}^{32}$. KDF is the key derivation function, where $\text{KDF} : \mathcal{B}^* \longrightarrow \mathcal{B}^{32}$. gen is the function dedicated to generating a pseudorandom matrix $\mathbf{A} \in R_q^{k \times k}$ from a seed ρ. The communication depicted in Figures 1 and 2 is as follows. Alice first randomly selects a seed d, hashing it to get the pair (ρ, σ). From σ, she generates two vector $\mathbf{s}$ and $\mathbf{e}$, serving as the secret key sk and a noise component, respectively. From ρ, she generates matrix $\mathbf{A}$, and then she computes $\mathbf{t}$ from $\mathbf{s}$, $\mathbf{e}$, and $\mathbf{A}$. Alice sends the public key pk, which is a pair of $\mathbf{t}$ and ρ, to

KEM.KeyGen()
$z \leftarrow B^{32}$
(pk, s) = PKE.KeyGen()
$sk = (s||pk||H(pk)||z)$
return (pk, sk)

$\xrightarrow{pk}$

KEM.Enc(pk)
$m_0 \leftarrow B^{32}$
$m = H(m_0)$
$(\bar{K}, r) = G(m||H(pk))$
c = PKE.Enc(pk, m; r)
K = KDF($\bar{K}$,$H(c)$)
return (c, K)

$\xleftarrow{c}$

KEM.Dec(c, $sk = (s||pk||H(pk)||z)$)
m'=PKE.Dec(c, s)
$(\bar{K}', r') = G(m'||H(pk))$
c' = PKE.Enc (pk, m'; r')
if $c = c'$
then return K = KDF($\bar{K}'$, $H(c)$)
else return K = KDF(z, $H(c)$)

Figure 1: Kyber.KEM.

PKE.KeyGen()
$d \leftarrow B^{32}$
$(\rho, \sigma) = G(d)$
$R_q^{k\times k} \ni$ A = gen(ρ)
$R_q^k \ni$ s, e $\leftarrow$ sampleCBD(σ)
t = As + e
return $(pk = (t||\rho), s)$

PKE.Enc(pk, m; r)
$(t||\rho) = pk$
$R_q^{k\times k} \ni$ A = gen(ρ)
$R_q^k \ni$ r, e_1 $\leftarrow$ sampleCBD(r)
$R_q \ni e_2$ $\leftarrow$ sampleCBD(r)
$u = A^T r + e_1$
$v = t^T r + e_2 + \text{Decompress}_q(m, 1)$
$c_1 = \text{Compress}_q(u, d_u)$
$c_2 = \text{Compress}_q(v, d_v)$
return $c = (c_1||c_2)$

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

Figure 2: Kyber.PKE.

Bob. When receiving pk, Bob performs the step KEM.Enc (i.e., Encaps) as depicted in Figure 1. He randomly selects an m_0, hashes it, and passes the outputs to the procedure PKE.Enc (depicted in Figure 2). The obtained ciphertext c, which is a pair of c_1 and c_2, is sent back to Alice. When receiving c, Alice performs the step KEM.Dec (i.e., Decaps). She computes c' by employing the procedures PKE.Dec and PKE.Enc. c' will equal c with an overwhelming probability, implying that m' on Alice's side is equal to m on Bob's side, and they can derive the same key K. Note that all calculations in the two figures are computed over $\mathbb{Z}_q[X]/(X^n + 1)$. Hereinafter, let us call the first message sending the public key pk from Alice to Bob the public key message, and the second message sending the ciphertext c from Bob to Alice the ciphertext message.

Kyber also employs two functions Compress and Decompress. Let $x \in \mathbb{Z}_q$ and $d < \lfloor \log_2 q \rceil$, the two functions are defined by the following equations:

$$\text{Compress}_q(x, d) = \left\lfloor \left(2^d/q\right) \cdot x \right\rceil \bmod 2^d, \quad (1)$$

$$\text{Decompress}_q(x, d) = \left\lfloor \left(q/2^d\right) \cdot x \right\rceil. \quad (2)$$

KEM.KeyGen()
$(seed_A, b, s)$ = PKE.KeyGen()
$pk = (seed_A, b)$
$pkh = F(pk)$
$z \leftarrow B^{32}$
$sk = (z, pkh, pk, s)$
return (pk, sk)

$\xrightarrow{pk}$

KEM.Enc(pk)
$m \leftarrow B^{32}$
$(\hat{K}, r) = G(F(pk), m)$
c = PKE.Enc(pk, m; r)
$K = H(\hat{K}, c)$
return (c, K)

$\xleftarrow{c}$

KEM.Dec(c, sk)
m' = PKE.Dec(s, c)
$(\hat{K}', r') = G(pkh, m')$
c' = PKE.Enc(pk, m'; r')
if c = c′
then return $K = H(\hat{K}', c)$
else return $K = H(z, c)$

Figure 3: Saber.KEM.

When Compress_q and Decompress_q are used with $x \in R_q$ or $\mathbf{x} \in R_q^k$, they are applied to each coefficient individually.

Note that in the latest submission document to NIST [11], the definition of Kyber employs two other functions Encode and Decode to serialize/deserialize polynomials/byte arrays. Besides, number-theoretic transform (NTT) is used to efficiently calculate multiplications in R_q. Because implementation or performance is out of the scope of the present paper, we omit those concepts for the sake of simplicity.

The procedure gen(ρ) in Figure 2 to generate the matrix $\mathbf{A}$ is deterministic. That means if two principals share the same random seed ρ, then they can agreeably derive the same matrix $\mathbf{A}$. The procedure sampleCBD to sample noise (or error) components (e.g., $\mathbf{e}$, $\mathbf{e_1}$, and e_2) takes as input a random seed (e.g., ρ and r) and returns as output a polynomial whose coefficients are close to a centered binomial distribution (to sample a vector, e.g., $\mathbf{r}$ and $\mathbf{e_1}$, the procedure is called multiple times). In other words, the coefficients are mostly close to 0 and their absolute values are never greater than a specific small threshold (can be either 5, 4, or 3, depending on the desired level of security).

3.2. Saber. Figure 3 depicts the triple algorithms (KeyGen, Encaps, and Decaps) of Saber.KEM, which employ three algorithms (KeyGen, Enc, and Dec) of Saber. PKE is depicted in Figure 4. $\mathscr{F}$, G, and $\mathscr{H}$ are three hash functions. β_μ denotes a centered binomial distribution. $\ll$ and $\gg$ are bitwise shift operators (when they are used with polynomials and matrices, the computation is performed on each coefficient). Different from Kyber where the security is achieved basically by adding a random noise sampled from an error distribution to make the public key or the ciphertext, Saber relies on the rounding functionality of the bitwise shift operators, which gains good efficiency and significantly reduces the amount of randomness required. Note that $\mathbf{h}$, h_1, and h_2 in the two figures are constants of vectors and polynomials; while q ($= 2^{\epsilon_q}$), p ($= 2^{\epsilon_p}$), T ($= 2^{\epsilon_T}$), and μ are integers receiving different values on different security levels.

PKE.KeyGen()
$seed_A \leftarrow B^{32}$
$A = \text{gen}(seed_A) \in R_q^{l \times l}$
$r \leftarrow B^{32}$
$s \leftarrow \beta_\mu(R_q^{l \times 1}; r)$
$b = ((A^T s + h) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
return $(pk = (seed_A, b), s)$

PKE.Enc($pk = (seed_A, b)$, m; r)
$A = \text{gen}(seed_A) \in R_q^{l \times l}$
$s' \leftarrow \beta_\mu(R_q^{l \times 1}; r)$
$b' = ((As' + h) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
$v' = b^T(s' \bmod p) \in Rp$
$c_m = (v' + h_1 - 2^{\epsilon_p - 1} m \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
return $c = (c_m, b')$

PKE.Dec(s, $c = (c_m, b')$)
$v = b'^T(s \bmod p) \in R_p$
$m' = ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$
return m'

Figure 4: Saber.PKE.

KeyGen()
$s_1 \leftarrow R_q^k$
$x_1 = \lfloor \frac{p}{q} A s_1 \rceil \in R_q^k$
return $(pk = x_1, sk = s_1)$

$\xrightarrow{pk}$

Enc($pk = x_1$)
$s_2 \leftarrow R_q^k$
$x_2 = \lfloor \frac{p}{q} A s_2 \rceil \in R_q^k$
$k_2 = x_1 s_2 \bmod p$
$w_2 = \text{Hiho}(k_2)$
$sk_2 = \lfloor \frac{p}{q} k_2 \rceil \bmod 2$
return $(c = (x_2, w_2), K = sk_2)$

$\xleftarrow{c}$

Dec($c = (x_2, w_2)$, $sk = s_1$)
$k_1 = x_2 s_1 \bmod p$
$sk_1 = \lfloor \frac{p}{q}(k_1 - w_2 \frac{p}{4} + \frac{p}{8}) \rceil \bmod 2$
return $K = sk_1$

Figure 5: SK-MLWR.

To see their possible values as well as other detailed information about the mechanism, we refer readers to the article [5] and its latest submission document to NIST [12].

3.3. SK-MLWR. Figure 5 depicts the triple algorithms (`KeyGen`, `Encaps`, and `Decaps`) of SK-MLWR. In this mechanism, p and q are integer constants, while $\mathbf{A}$ is the public matrix known by everyone. Let $x \in \mathbb{Z}_p$, then the additional information function $\text{HiHo} : \mathbb{Z}_p \longrightarrow \mathbb{Z}_2$ is defined as follows:

$$\text{HiHo}(x) = \left\lfloor 4\frac{x}{p} \right\rceil \bmod 2. \tag{3}$$

The mechanism execution is well-explained in Figure 5, and thus we skip explaining it in detail. We again refer readers to the article [6] for the information.

3.4. Security Property. The security of Kyber, Saber, and SK-MLWR [4–6] is defined in terms of a *game* between a *challenger* and an *adversary*, where the authors gave computational proofs that no polynomial-time adversary can obtain a nonnegligible distinguishing a real shared secret key established by those KEMs and a random value. The adversary is said to have a nonnegligible advantage if they can succeed in that distinguishment challenge with a probability significantly greater than ½. Note that those proofs were achieved manually, but not verified by the computers.

This paper takes into consideration the secrecy property of shared secret keys, which states that: if Alice in her belief obtains a shared secret key k with Bob through Kyber, Saber, or SK-MLWR, then Eve, a third party, is unable to learn k. Probability is not taken into account in our analysis as it is not a computational-based approach.

4. A Base Specification

The three lattice-based KEMs share many common parts in their designs, such as polynomials, vectors, matrices, and message exchange patterns. Thus, as the first step, we modeled these common parts, combining them into a specification, which we called a base specification. Then, from the base specification, to model each of the three KEMs, we only need to model some additional parts as well as the mechanism execution. This section presents how to model those common parts in Maude to construct that base specification.

4.1. Modeling Polynomials, Vectors, and Matrices. We first define Maude module `POLYNOMIAL` which is dedicated to specifying polynomials. The definition of the module starts with:

```
fmod POLYNOMIAL is
  pr INT .
  sort Poly . subsort Int < Poly .
```

where **`fmod`** stands for a functional module, and **`pr`** `INT` says that the module imports the Maude pre-defined module `INT`, which specifies integers. Sort (type) `Poly` represents polynomials. The notation **`subsort`** `Int < Poly` says that any integer is also a polynomial, where `Int` is the sort of integers defined in the module `INT`.

We then declare four operators `p+`, `p*`, `p-`, and `neg`, where the first three ones, respectively, denote the addition, multiplication, and subtraction between two polynomials, while the last one denotes the negation of a polynomial. They are as follows:

```
op _p+_ : Poly Poly -> Poly [ctor assoc
comm prec 33] .
op _p*_ : Poly Poly -> Poly [ctor assoc
comm prec 31] .
op _p-_ : Poly Poly -> Poly [prec 33] .
op neg_ : Poly -> Poly [ctor] .
```

where `ctor`, `assoc`, and `comm` are three equational attributes, saying that `p+`, `p*`, and `neg` are constructors of the

sort Poly; p+ and p* are associative and commutative. prec 33 says that the precedence of p+ and p− is lower than that of p* (the higher number, the less precedence). Some basic properties of these operators are specified by means of equations as follows:

```
--- P, P', and P are variables of the sort Poly
vars P P' P'' : Poly .
eq P p+ 0 = P .
eq P p* 0 = 0 .
eq P p* 1 = P .
eq P p* (P' p+ P'') = (P p* P') p+ (P p* P'') .
```

The first equation states that any polynomial P plus 0 is P itself. The next two equations can be understood likewise. The last equation specifies the distributive property over addition and multiplication. Similarly, some properties of p− and neg are defined, and we complete definition of the module POLYNOMIAL:

```
  eq P p- P' = P p+ neg(P') .
  eq P p+ neg(P') = 0 .
  eq neg(neg(P)) = P .
  eq neg(P p+ P') = neg(P) p+ neg(P') .
endfm
```

In the same manner, we define module VECTOR with the sort Vector to specify polynomial vectors. In this module, tp is defined as a transpose operator (transposes a column vector by default to a row vector). Two operators v+ and dot are introduced representing the addition and inner product of two polynomial vectors. Note that our specification considers only two vectors with the same dimension. Their declarations and some basic properties are as follows:

```
sort Vector . subsort Poly < Vector .
op tp : Vector -> Vector .
op _v+_ : Vector Vector -> Vector [assoc comm prec 33] .
op _dot_ : Vector Vector -> Poly [prec 31] .
vars V V' V'' : Vector .
eq tp(tp(V)) = V .
eq tp(V v+ V') = tp(V) v+ tp(V') .
eq (V v+ V') dot V'' = (V dot V'') p+ (V' dot V'') .
eq V'' dot (V v+ V') = (V'' dot V) p+ (V'' dot V') .
```

We continuously define module MATRIX with the sort Matrix to specify polynomial matrices. In this module, in addition to the transpose operator tp, we introduce the operator m* denoting the multiplication of a matrix and a vector:

```
op _m*_ : Matrix Vector -> Vector [prec 31] .
vars M : Matrix .
eq tp(tp(M)) = M .
eq tp(M m* V) dot V' = tp(V) dot (tp(M) m* V') .
```

Note that our specification considers only square matrices and the multiplication is defined between only matrices and vectors which have proper dimensions. It is unnecessary to consider the other cases since there is no such an operation in the three KEMs.

The procedures to generate pseudorandom matrices and vectors from random seeds used in Kyber and Saber are specified by the following operators:

```
--- generate a matrix from a random seed
op gen-A : Poly -> Matrix .
--- used in Kyber and Saber
op sample-s : Poly -> Vector .
--- used in Kyber, initiator side
op sample-e : Poly -> Vector .
--- used in Kyber, responder side
op sample-r : Poly -> Vector .
op sample-e1 : Poly -> Vector .
op sample-e2 : Poly -> Poly .
```

gen-A represents the procedure gen, taking as input a random seed (ρ in Kyber or $\text{seed}_{\mathbf{A}}$ in Saber) and outputting matrix $\mathbf{A}$. sample-s specifies the procedures generating the secret vectors $\mathbf{s}$ and $\mathbf{s}'$ from a random seed (σ in Kyber or r in Saber). sample-e represents the procedure producing the noise vector $\mathbf{e}$, which is used in the initiator side of Kyber (or precisely, in the step KeyGen). Similarly, sample-r, sample-e1, sample-e2 produce $\mathbf{r}$, $\mathbf{e}_1$, and e_2, respectively, used in Kyber.PKE.Encaps.

Recall that the aforementioned procedures produce vectors and polynomials whose coefficients are small (in comparison with p and q). To specify that property, the following predicate of vectors and polynomials is defined:

```
op isSmall? : Vector -> Bool .
eq isSmall?(sample-s(P)) = true .
eq isSmall?(sample-e(P)) = true .
eq isSmall?(sample-r(P)) = true .
eq isSmall?(sample-e1(P)) = true .
eq isSmall?(sample-e2(P)) = true .
eq isSmall?(tp(V)) = isSmall?(V) .
ceq isSmall?(V v+ V') = true
 if (isSmall?(V) and isSmall?(V')) .
ceq isSmall?(V dot V') = true
 if (isSmall?(V) and isSmall?(V')) .
ceq isSmall?(P p+ P') = true
 if (isSmall?(P) and isSmall?(P')) .
ceq isSmall?(neg(P)) = true if isSmall?(P) .
```

isSmall?(P) (or isSmall?(V)) is true if all of its coefficients are small. The first five equations indicate that the

sampling procedures `sample-s`, `sample-e`, … all return vectors or polynomials whose coefficients are small. The last five equations specify some properties of this predicate. For instance, if the coefficients of two vectors `V` and `V'` are small, then the coefficients of their addition are also small.

4.2. Modeling Messages Exchanged and Common Observable Components. We introduce sort `Msg` with two operators `msg1` and `msg2` representing public key and ciphertext messages, respectively, as follows:

```
op msg1 : Prin Prin Prin PVPair -> Msg [ctor] .
op msg2 : Prin Prin Prin PVPair -> Msg [ctor] .
```

where `Prin` is the sort of principals. The first, second, and third arguments of `msg1` and `msg2` are the actual creator, the seeming sender, and the receiver of the corresponding message. For example, when the first, second, and third arguments are an intruder, Alice, and Bob, respectively, then that intruder created the message, impersonating Alice to send the message to Bob. Note that this first argument cannot be seen by the receiver, but it is meta-information that is only available to the outside observer.

The last argument, namely `PVPair`, is the sort of tuples of vectors and polynomials, whose definition is as follows:

```
op _&&_ : Vector Poly -> PVPair [ctor] .
```

This argument carries actual data exchanged between two principals, i.e., public key pk or ciphertext *c*. For example, in Kyber, the public key consists of a vector $\mathbf{t}$ and a random seed ρ, which can be expressed as a term of the sort `PVPair` (e.g., $\mathbf{t}$ && ρ).

Recall that we model each KEM as a state machine in Maude, where each state is represented by a braced AC-collection of observable components (i.e., name-value pairs). For all the three KEMs, we use some common observable components as follows:

(i) (`prins` : *prs*)—*prs* are all principals participating in the mechanisms. In each experiment reported in this paper with each of the three KEMs, participants participating are fixed as three, including two honest ones, namely `alice` and `bob`, and an intruder, namely `eve`. The intruder is modeled based on the Dolev and Yao [8] model, which gives them capabilities of intercepting, modifying, faking messages, and impersonating other protocol principals.

(ii) (`nw` : *msgs*)—*msgs* is the set of messages exchanged by all principals in the network.

(iii) (`keys[p]` : *ks*)—*ks* is the set of the shared keys that principal `p` has established with others. Each entry of *ks* is in the form of either:

(1) `Initiator(K, q)`: the shared key `K` is established with principal `q`, where `p` is the initiator (i.e., who starts the communication), or

(2) `Responder(K, q)`: the shared key `K` is established with principal `q`, where `p` is the responder.

(iv) (`glean-keys` : *gks*)—*gks* is the set of shared keys learned by the intruder `eve`.

To complete the base specification, the operator `KDF` is declared representing the key derivation function KDF (used in Kyber), and the operators `H` and `G` are introduced representing the hash functions $\mathscr{H}$ and $\mathscr{G}$ (used in Kyber and Saber).

4.3. Intruder Capabilities. For all of the three KEMs, we suppose the presence of an intruder, namely, `eve`, who can control all the network communications. We give the intruder the following concrete capabilities:

(1) If `A` sends a public key message to `B`, `eve` can intercept that message, fake a new message with a new public key, and pretend `A` to send it to `B`.

(2) `eve` can randomly generate values to be used as random seeds d, m_0, and so forth.

(3) Once a public key message sent by `A` to `B` is intercepted and forged with a new message, `eve` can continuously impersonate `B`, construct by themself a ciphertext *c*, send it back to `A` as a reply, and compute a shared secret key (with `A`).

(4) When `B` replies back to `A` with a ciphertext message after receiving a public key message apparently sent from `A`, `eve` can intercept that ciphertext message, and compute a shared secret key (with `B`) from the ciphertext received.

5. Model Checking Kyber

This section reports the complete Kyber specification expanded from the base one, followed by its analysis result. Saber's and SK-MLWR's results will be reported in the next section.

5.1. Complete Kyber Formal Specification. Extending the base observable components listed in the last section, we introduce the following new observable components in the formal specification of Kyber:

(i) (`d[p]` : d_0)—the random seed d (used in Figure 2) of principal `p` receives d_0 as a value.

(ii) (`m[p]` : m_0)—the random seed m_0 (used in Figure 1) of principal `p` receives m_0 as a value.

(iii) (`rd-d` : rds)—rds is a list of available values for the random seed d. Whenever a principal needs to randomly select a value for d, the top entry of rds is removed and returned to the principal. Thus, the uniqueness is guaranteed provided that, initially, entries in rds are different from each other. List structure is used for rds, but not a set, because it helps to reduce the size of the state space so that

the time taken for the reachability analysis can be shortened.

(iv) (rd-m : *rms*)—*rms* is a list of available values for the random seed m_0.

(v) (ds : *eds*)—*eds* is the set of values for the random seed *d* that are available to the intruder (either learned or randomly selected).

(vi) (ms : *ems*)—*ems* is the set of values for the random seed m_0 that are available to the intruder (similarly, either learned or randomly selected).

Initial states are then represented by a constant init, which is defined as follows:

```
eq init =
{(prins: (alice bob eve)) (nw: empty) (rd-
d: (d1, d2))
(d[alice]: 0) (d[bob]: 0) (m[alice]: 0) (m
[bob]: 0)
(rd-m: (m1, m2)) (keys[alice]: empty)
(keys[bob]: empty)
(glean-keys: empty) (ds: empty) (ms:
empty)} .
```

5.1.1. Modeling Kyber Execution. We specify three rewrite rules keygen, encaps, and decaps, modeling the corresponding three algorithms of Kyber. In other words, the three rewrite rules model (1) Alice generates a public key message, (2) Bob accepted the public key message and generates a ciphertext message, and (3) Alice accepts the ciphertext message, respectively. The rewrite rule keygen is defined as follows:

```
vars A B C : Prin .
vars PS : PrinSet . —principal sets
vars PoL : ListPoly . —polynomial lists
vars MS : Network . —message sets
vars D M M0 P Rho M' V CV V' Rseed Sig : Poly .
crl [keygen] :
    {(prins: (A; B; PS)) (rd-d: (D, PoL))
    (d[A]: P)
  (nw: MS) OCs}
=> {(prins: (A; B; PS)) (rd-d: PoL) (d
   [A]: D)
      (nw: (msg1(A, A,B, (gen-A(Rho) m*
      sample-s(Sig) v+
        sample-e(Sig)) && Rho); MS)) OCs}
if Rho := 1st(G(D)) /\
   Sig := 2nd(G(D)) .
```

OCs is a variable of observable component collections. Note that the output of G(D) is a polynomial pair, where 1st and 2nd are the projection operators returning the first and the second components, respectively. Recall that && is the constructor of tuples of vectors and polynomials; gen-A represents the function gen, generating the pseudorandom matrix **A**; sample-e and sample-s produces the vectors **e** and **s**, respectively. The rewrite rule says that when there exists a polynomial D in rd-d, A selects it as a value for the random seed *d*, computes a public/secret key pair exactly following the algorithm KeyGen of the mechanism, and sends the public key to B. Together with that, d[A] is updated to D, and D is removed from rd-d. Note that the operators, such as m*, v+, and p+, are implicitly understood to be computed over R_q, and then we do not explicitly specify the modulo operation.

The rewrite rule encap is defined as follows:

```
vars KS : KeySet .
vars T U CU U' : Vector .
vars Kr Kr2 : PolyPair . —polynomial pairs
crl [encaps] :
    {(rd-m: (M0, PoL)) (m[B]: P) (keys
    [B]: KS)
     (nw: (msg1(C, A,B, T && Rho); MS))
     OCs}
=> {(rd-m: PoL) (m[B]: M0) (keys[B]:
     (KS; responder(KDF(1st(Kr) || H(CU
     && CV)), A)))
     (nw: (msg1(C, A,B, T && Rho);
          msg2(B, B,A, CU && CV); MS))
          OCs}
if M := H(M0) /\
  Kr := G(M || H(T && Rho)) /\
  CU := enc-u(T, Rho, M, 2nd(Kr)) /\
  CV := enc-v(T, Rho, M, 2nd(Kr)) .
```

where enc-u and enc-v compute c_1 and c_2 in PKE. $\mathrm{Enc}(pk, m; r)$ in Figure 2, respectively. They are defined as follows:

```
eq enc-u(T, Rho, M, Rseed) =
  compr(tp(gen-A(Rho))   m*   sample-r
  (Rseed) v+sample-e1(Rseed), du) .
eq enc-v(T, Rho, M, Rseed) =
  compr(tp(T)  dot  sample-r(Rseed)  p
  +sample-e2(Rseed) p+
    decompr(M, 1), dv) .
```

compr and decompr denote the functions $\mathrm{Compress}_q$ and $\mathrm{Decompress}_q$, respectively, while du and dv respectively denote the constants *du* and *dv*.

The rewrite rule encaps says that when B receives a public key message seemingly sent from A, B first selects a random M0, from that computes two components CU and CV of the ciphertext *c*, and sends the ciphertext back to A. Together with that, B also computes the shared key with A, and the state of the message B received is updated to

replied. One delicate point is that the message B received is in the form of msg1(C, A,B,...), which means that the creator of the message is actually C, which may or may not be A. In particular, when C is the intruder (which is unable for B to recognize), then the message is faked by the intruder but not sent by A.

The rewrite rule decaps is defined as follows:

```
crl [decaps] :
    {(d[A] : D) (keys[A] : KS)
     (nw: (msg1(A, A,B, T && Rho);
          msg2(C, B,A, CU && CV); MS))
          OCs}
=> {(d[A] : D) (keys[A] :
        (KS; initiator(KDF(1st(Kr2) || H
        (CU && CV)), B)))
     (nw: (msg1(A, A,B, T && Rho);
          msg2(C, B,A, CU && CV); MS))
          OCs}
if Sig := 2nd(G(D)) /\
  U' := decompr(CU, du) /\
  V' := decompr(CV, dv) /\
  M' := compr(V' p- tp(sample-s(Sig)) dot
  U', 1) /\
  Kr2 := G(M' || H(T && Rho)) /\
  CU == enc-u(T, Rho,M', 2nd(Kr2)) /\
  CV == enc-v(T, Rho,M', 2nd(Kr2)) .
```

The rewrite rule says that when A has sent a public key message to B and there exists a message apparently replied from B with the ciphertext consisting of CU and CV, then A decompresses CU and CV, from that recovers m (on the Bob side), and computes the shared key with B. We repeat again that Kyber is assumed to be 0-*correct*, that is we only consider the case when Alice successfully recovers m, which happened with an overwhelming probability. To this end, the error tolerance gaps made by noise components must be silent, which is done by the following equation:

```
ceq compr(P p+decompr(M, 1),1) = M if
isSmall?(P) .
```

Let $x' = \text{Decompress}_q\big(\text{Compress}_q(x,d),d\big)$. From Equations (1) and (2), we have:

$$|x' - x \bmod q| \leq \left\lfloor \frac{q}{2^{d+1}} \right\rceil. \tag{4}$$

Using this inequality, $\text{Decompress}_q\big(\text{Compress}_q(v,dv),dv\big)$ and $\text{Decompress}_q\big(\text{Compress}_q(\mathbf{u},du),du\big)$ can be rewritten to $v+\epsilon_1$ and $\mathbf{u}+\epsilon_2$, respectively, where all coefficients of ϵ_1 and ϵ_2 are small in comparison with those of v and $\mathbf{u}$. In the specification, we specify ϵ_1 as epsilon1(v), ϵ_2 as epsilon2($\mathbf{u}$), and both epsilon1(v) & epsilon2($\mathbf{u}$) are "small". This is done by the following equations:

```
eq decompr(compr(V, dv),dv) = V p+epsilon1
(V) .
eq decompr(compr(U, du),du) = U v+epsilon2
(U) .
eq isSmall?(epsilon1(V)) = true .
eq isSmall?(epsilon2(U)) = true .
```

5.1.2. Checking the Specification. We check that the specification we have specified so far allows two principals successfully establish a shared key. This must be fulfilled, otherwise, anything we do after is completely meaningless. To this end, we define the following Maude **search** command:

```
search [1] in KYBER : init => *
  {(keys[alice] : initiator(K:Poly, bob))
   (keys[bob] : responder(K:Poly, alice))
   OCs} .
```

The command tries to find a state reachable from init such that both alice and bob obtain the shared key K:Poly. Maude found a solution for the command, meaning that two principals can successfully establish a shared key.

5.1.3. Modeling the Intruder. We turn to specify intruder capabilities. Recall that we assume the presence of an intruder, i.e., eve, with the capabilities mentioned in Section 4.3. The following rewrite rule specifies its capability (1):

```
vars PoS : PolySet .
crl [keygen-eve] :
    {(ds: (D; PoS))
     (nw: (msg1(A, A,B, TA && RhoA); MS))
     OCs}
=> {(ds: (D; PoS))
     (nw: (msg1(A, A,B, TA && RhoA);
          msg1(eve, A,B, (gen-A(Rho) m*
          sample-s(Sig) v+
             sample-e(Sig)  &&  Rho);
             MS)) OCs}
if Rho := 1st(G(D)) /\
   Sig := 2nd(G(D)) .
```

From an available value D for the random seed d, eve computes a new public key and impersonates A to send the faking message to B (msg1(eve, A,B,...)). The random value D cannot be collected from the network, but eve can only construct it by randomly selecting a new value. Similarly, the only way in which eve can construct values for the random seed m is by randomly selecting a new value. This has been mentioned as capability (2), which is specified in Maude as follows:

```
rl [build-ds] : {(rd-d: (P, PoL)) (ds:
PoS) OCs}
=> {(rd-d: PoL) (ds: (P; PoS)) OCs} .
```

```
rl [build-ms] : {(rd-m: (P, PoL)) (ms:
PoS) OCs}
=> {(rd-m: PoL) (ms: (P; PoS)) OCs} .
```

Intruder capability (3) is specified by the following rewrite rule:

```
crl [encaps-eve] :
    {(ms: (M0; PoS)) (glean-keys: KS)
     (nw: (msg1(A, A,B, TA && RhoA); MS))
     OCs}
=> {(ms: (M0; PoS)) (glean-keys:
       (responder(KDF(1st(Kr) || H(CU
       && CV)), A); KS))
     (nw: (msg1(A, A,B, TA && RhoA);
          msg2(eve, B,A, CU && CV); MS))
          OCs}
if M := H(M0) /\
   Kr := G(M || H(TA && RhoA)) /\
   CU := enc-u(TA, RhoA, M, 2nd(Kr)) /\
   CV := enc-v(TA, RhoA, M, 2nd(Kr)) .
```

The last rewrite rule, namely decaps-eve, specifies intruder capability (4):

```
crl [decaps-eve] :
    {(ds: (D; PoS)) (glean-keys: KS)
     (nw: (msg1(eve, A,B, T && Rho);
          msg2(B, B,A, CUB && CVB); MS))
          OCs}
=> {(ds: (D; PoS)) (glean-keys:
      (initiator(KDF(1st(Kr2) || H(CUB
      && CVB)), B); KS))
   (nw: (msg1(eve, A,B, T && Rho);
        msg2(B, B,A, CUB && CVB); MS))
        OCs}
if Sig := 2nd(G(D)) /\
   U' := decompr(CUB, du) /\
   V' := decompr(CVB, dv) /\
   M' := compr(V' p- tp(sample-s(Sig)) dot
   U', 1) /\
   Kr2 := G(M' || H(T && Rho)) /\
   CUB == enc-u(T, Rho,M', 2nd(Kr2)) /\
   CVB == enc-v(T, Rho,M', 2nd(Kr2)) .
```

5.2. Model Checking and Man-in-the-Middle Attack. The formal specification of Kyber in Maude is completed, and now we are ready to perform analysis to check the property mentioned in Section 3. To this end, we introduce the following Maude **search** command:

```
search [1] in KYBER : init =>*
  {(keys[alice] : initiator(K:Poly, bob))
   (keys[bob] : responder(K':Poly, alice))
   (glean-keys: (responder(K:Poly, alice);
                 initiator(K':Poly, bob);
                 KS)) OCs} .
```

K and K′ may or may not be equal. The **search** command tries to find a state reachable from init such that: alice in her belief obtains the shared key K with bob, bob in his belief obtains the shared key K′ with alice, and eve learned both K and K′. Maude found a counterexample after about 7 min and 43 s on a computing server that carries 384 GB of memory and 16 cores 2.8 GHz microprocessor. This kind of vulnerability belongs to MITM attacks. Figure 6 depicts how this attack happens on Kyber, which is visualized from the path leading to the counterexample Maude returned. There are mainly six steps as follows:

Step 1: Alice initializes a key exchange with Bob by performing the procedure KEM.KeyGen. She obtains a secret key sk, which is kept by her, and a public key pk, which is sent to Bob.

Step 2: Eve intercepts the public key message sent from Alice to Bob. She selects a random d_e to generate by herself a public/secret pair $(\mathrm{pk}_e, \mathrm{sk}_e)$ following the procedure KEM.KeyGen. She impersonates Alice to send pk_e to Bob.

Step 3: Bob receives the public key message containing pk_e, believing it is from Alice. Subsequently, he selects a random m_0 to perform KEM.Enc, obtaining a ciphertext c and a shared key K_b. He keeps the key K_b, which he believes that it is the shared key established by him and Alice. He sends the ciphertext c back to Alice as a response to the public key message.

Step 4: Eve once again intercepts the ciphertext message sent from Bob to Alice. Afterward, she selects a random m_{e0}, performs KEM.Enc on the inputs pk and m_{e0}, and obtains a different ciphertext & shared key pair, namely c_e and K_a. She impersonates Bob to send the ciphertext c_e back to Alice as a reply to the very first public key message.

Step 5: Alice receives the ciphertext message containing the ciphertext c_e, believing it is from Bob. Then, she performs KEM.Dec on the inputs c_e and sk, obtaining the shared key K_a. She believes that the shared key K_a is established by her and Bob.

Step 6: Finally, Eve performs KEM. Dec on the inputs c and sk_e, obtaining the shared key K_b.

6. Model Checking Saber and SK-MLWR

In the similar way, we model Saber and SK-MLWR in Maude and then conduct the similar model checking experiments. The same MITM attacks are found by Maude. In this section, we mainly focus on presenting how to complete the Maude

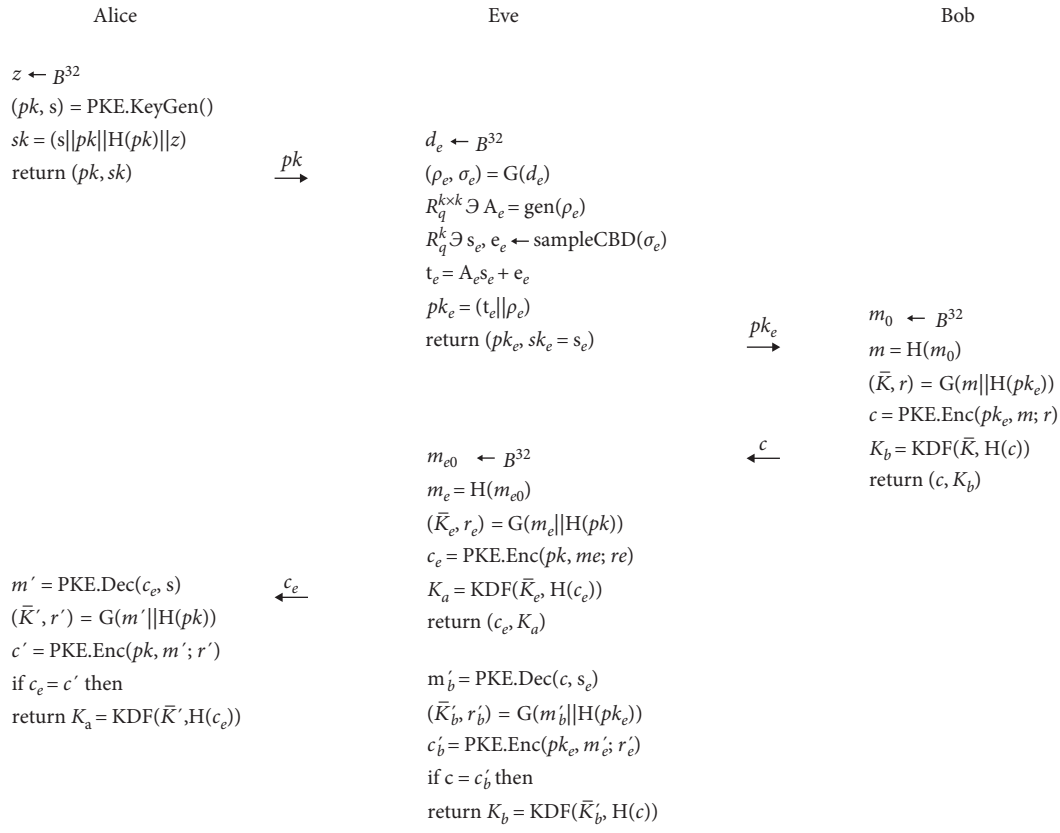

FIGURE 6: A visualization of the counterexample.

formal specifications of the two mechanisms from the base specification presented in Section 4.

6.1. Complete Saber Formal Specification in Maude. We first extend the modules `POLYNOMIAL` and `VECTOR` in the base specification, adding the modulo operator and the bitwise shift (precisely, right shift) operator:

```
– modulo operator
op _md_ : Poly NzNat -> Poly [prec 31] .
op _md_ : Vector NzNat -> Vector [prec 31] .
– bitwise right shift operator
op shiftR : Poly Int -> Poly .
op shiftR : Vector Int -> Vector .
```

Note that `NzNat` is the sort of non-zero natural numbers (pre-defined in Maude). After that, some basic properties related to them are defined:

```
eq (P p+ (P' md K)) md K = (P p + P') md K .
eq neg(P md K) = neg(P) md K .
```

The observable components are updated by adding the following new ones:

(i) (`seed[p] : sd`)—the value of the random seed seed_A (used in Figure 4) of principal `p` is *sd*.
(ii) (`r[p] : r0`)—the value of the random seed *r* (used in Figure 4) of principal `p` is r_0;
(iii) (`m[p] : m0`)—the value of the random seed *m* (used in Figure 3) of principal `p` is m_0;
(iv) (`rd-seed : rd1`)—rd_1 is a list of available values for the random seed seed_A.
(v) (`rd-r : rd2`)—rd_2 is a list of available values for the random seed *r*.
(vi) (`rd-m : rd3`)—rd_3 is a list of available values for the random seed *m*.
(vii) (`seeds : se`)—*se* is the set of values for the random seed seed_A that are available to the intruder (either learned or randomly selected).
(viii) (`rs : re`)—*re* is the set of values for the random seed *r* that are available to the intruder.
(ix) (`ms : me`)—*me* is the set of values for the random seed *m* that are available to the intruder.

We define three rewrite rules `keygen`, `encaps`, and `decaps` to model the mechanism execution; and three rewrite rules `keygen-eve`, `encaps-eve`, and `decaps-eve` to model

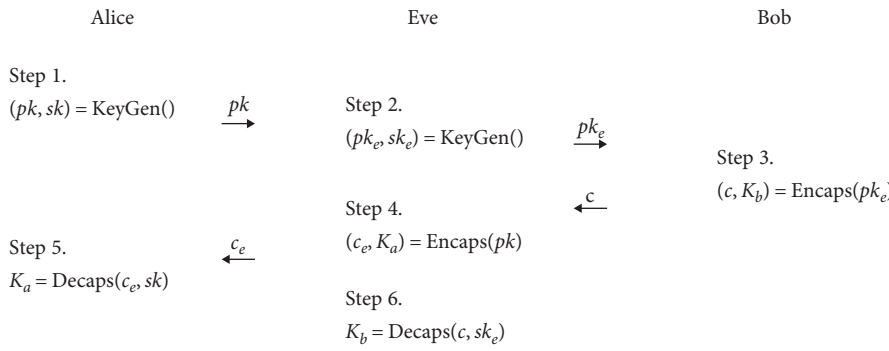

FIGURE 7: High-level representation of attacks.

the intruder capabilities. We show here the definition of the rewrite rule `keygen`. For the others, readers can find their definitions on the webpage mentioned in Section 1. The definition is as follows:

```
var MA : Matrix .
vars S VB : Vector .
var SD : Poly .
var PoL2 : ListPoly .
crl [keygen] :
    {(prins: (A; B; PS)) (r[A]: P') (rd-r:
    (R, PoL2))
     (seed[A]: P) (rd-seed: (SD, PoL))
     (nw: MS) OCs}
=> {(prins: (A; B; PS)) (r[A]: R) (rd-r:
   PoL2)
     (seed[A]: SD) (rd-seed: PoL)
     (nw: (MS; msg1(A, A,B, VB && SD)))
    OCs}
if MA :=gen-A(SD) /\
   S :=sample-s(R) /\
   VB :=shiftR((tp(MA) m* S v+h) md q,
   esq - esp) .
```

Note that `esp` and `esq`, respectively, denote the integers ϵ_p and ϵ_q. The rewrite rule says that the random seeds `SD` in `rd-seed` and `R` in `rd-r` are selected by principal `A` as the values for the random seed $\text{seed}_{\mathbf{A}}$ and r, respectively, from those `A` computes matrix `MA` (i.e., $\mathbf{A}$) and secret vector `S` (i.e., $\mathbf{s}$). `A` then computes the vector `VB` (i.e., $\mathbf{b}$). Concatenation of the vector and the random seed `SD` forms the public key, which is then sent to `B` through a public key message.

6.2. Complete SK-MLWR Formal Specification in Maude. With the SK-MLWR case study, we define an additional module, namely `FRACTION`, that is dedicated to model fractions. In this module, a fraction is specified by the infix operator `_/_`, whose two input arguments are the sort of integers. Similar to the Saber case study, the modulo operation is added into the modules `POLYNOMIAL` and `VECTOR`. With the module `VECTOR`, we also define two operators: `round` denoting the rounding operation ($\lfloor \cdot \rceil$), and `v*` denoting the multiplication between a fraction and a vector. With the former, we assume that the rounding of a vector is the vector itself, which is specified by the following equation: `eq round(V:Vector) =V:Vector .`

The observable components are extending from the base ones by adding the following:

(i) (`s[p] : sp`)—the value of $\mathbf{s}_1$ or $\mathbf{s}_2$ (used in Figure 5) of principal `p` is *sp*.
(ii) (`rd-s : rs`)—*rs* is a list of available values for $\mathbf{s}_1$ and $\mathbf{s}_2$.
(iii) (`ss : ss`)—*ss* is the set of values for $\mathbf{s}_1$ and $\mathbf{s}_2$ that are available to the intruder.

The following is the definition of the rewrite rule `keygen` (for the others, we again refer readers to the webpage mentioned in Section 1):

```
var VL : ListVector . — vector lists
crl [keygen] :
    {(prins: (A; B; PS)) (s[A]: V) (rd-s:
    (S, VL))
     (nw: MS) OCs}
=> {(prins: (A; B; PS)) (s[A]: S) (rd-s:
   VL)
     (nw: (MS; msg1(A, A,B, X1:Vector)))
    OCs}
if X1:Vector :=round((p / q) v* ((mA m* S)
md q)) .
```

Note that `mA` denotes the public polynomial matrix $\mathbf{A}$. The rewrite rule says that the random vector `S` in `rd-s` is selected by principal `A` as a secret key, and from `S`, `A` computes the public key `X1` to send it to `B` through a public key message.

6.2.1. Model Checking with the Two KEMs. Once complete the formal specifications, we can straightforwardly conduct model-checking experiments similar to what has been reported in Section 5.2. The same results, i.e., MITM attacks, are found. From a high-level point of view, all of the attacks found for the three KEMs can be graphically visualized as in Figure 7. According to the figure, Alice generates a public/

secret key pair pk and sk, trying to send pk to Bob. The public key message is intercepted by Eve. She generates by herself another public/secret key pair pk_e and sk_e (which are completely irrelevant to pk and sk), acting as Alice to send pk_e to Bob. Upon receiving that public key message, Bob executes `Encaps` on the public key received, getting a ciphertext c and a shared key K_b. Thinking that the public key message came from Alice, he sends back to Alice the ciphertext and keeps K_b as the shared key with Alice. Eve once again intercepts the ciphertext message sent from Bob. She replaces the original ciphertext with a new one, i.e., c_e (which is completely irrelevant to c), impersonating Bob to send it back to Alice. Upon receiving the faking ciphertext, Alice executes `Decaps`, obtaining the shared key K_a, which is also in Eve's possession. In last, from the ciphertext c got from Bob, Eve calculates the shared key K_b.

7. Remark

We leave some remarks on the model-checking experiments that have been conducted.

7.1. Finite Reachable State Space. The first remark is that the reachable state space of each model checking experiment is finite. Indeed, this claim can be proved, for example with the Kyber case study, by the following command:

```
search in KYBER : init => * {OCs} .
```

Because there is no constraint on the state pattern we are searching for, the command above tries to find all states reachable from `init`. Running this command, 2,685 solutions (or states) are returned after about 21 min and 37 s. If we want to prove an invariant property, this finite reachable state space property must be fulfilled, otherwise, the reachability analysis will never terminate. For example, let us consider the following property: whenever Alice and Bob both agreeably established a shared key, the intruder cannot learn that key. To check this property, we introduce the following two **`search`** commands:

```
search [1] in KYBER : init => *
  {(keys[alice] : initiator(K:Poly, bob))
   (keys[bob] : responder(K:Poly, alice))
   (glean-keys : (initiator(K:Poly, X:
   Prin); KS)) OCs} .
search [1] in KYBER : init => *
  {(keys[alice] : initiator(K:Poly, bob))
   (keys[bob] : responder(K:Poly, alice))
   (glean-keys : (responder(K:Poly, X:
   Prin); KS)) OCs} .
```

The two commands try to find a state in which Alice and Bob both agreeably established the shared key `K` and Eve already learned that key (note that `X:Prin` denotes arbitrary principal since we do not mind about this parameter). The key point is that because Maude returns no solution for either of the two commands, it must have already exhaustedly traveled all reachable states. In other words, if the state space is infinite, the search will never terminate.

In the following, we give an explanation of why the state space is finite in the Kyber case study (the two other ones can be explained similarly). Recall that each state is denoted as a braced AC-collection of the ten observable components as shown in Sections 4 and 5. The key point is that the number of possible values that each observable component (i.e., a name-value pair) can receive is finite. Indeed, it is true that:

(i) prs in (`prins` : *prs*) is always `(alice bob eve)` because no rewrite rule produces or consumes principals.

(ii) rds and rms in (`rd-d` : *rds*) and (`rd-m` : *rms*) never consist of more than `(d1, d2)` and `(m1, m2)`, respectively, because no rewrite rule produces new entries into these lists.

(iii) d_0 in (`d[p]` : d_0) can only be either `0`, `d1`, or `d2`. Similarly, m_0 in (`m[p]` : m_0) can only be either `0`, `m1`, or `m2`.

(iv) eds in (`ds` : eds) must be a subset of {`d1, d2`}, and then the numbers of possible values is finite. Similarly, the number of possible values of ems in (`ms` : ems) is finite.

(v) The number of possible values of *msgs* in (`nw` : *msgs*) is finite because:

  (1) The two rewrite rules `keygen` and `encaps` each produces a new message into the network while also consuming one element from rds and rms. Because rds and rms never consist of more than `(d1, d2)` and `(m1, m2)` as explained above, the two rewrite rules can only be applied finitely many times.

  (2) The rewrite rule `keygen-eve` produces a new public key message into the network, which can be applied infinite times to put the same message into the network again and again. However, because the network is modeled as a set of messages, putting a message that is already existing in the network into the network is meaningless, the network will remain unchanged.

  (3) Similarly, the rewrite rule `encaps-eve` cannot produce an infinite ciphertext messages into the network.

  (4) The other rewrite rules do not update the network.

(vi) Similarly, the number of possible values of *gks* in (`glean-keys` : *gks*) is finite because *gks* is a set, and so even if the two rewrite rules `encaps-eve` and `decaps-eve` apply infinite times with the same source state, *gks* cannot be infinite.

(vii) Similarly, the number of possible values of *ks* in (`keys[p]` : *ks*) is finite because: (1) the rewrite rule `encaps` consumes an element from *rms*; and (2) although the rewrite rule `decaps` can be applied infinite times with the same source state, *ks* remains finite because it is a set.

From what has been explained, it follows that the state space in this case is finite. Consequently, with any invariant model checking experiment, Maude will eventually return either some solution(s) or no solution after a finite time. Because the Maude execution process is fully automated, which is an advantage of the analysis approach, we can easily check other desired properties. Given a formal specification of the protocol under analysis, once we complete specifying the desired property, the analysis can be done automatically by Maude.

7.2. The Attack Found Is Not Novel. The attacks found in the three case studies can be said that are not novel attacks since the three KEMs are not equipped with any feature for dealing with authentication. That is the reason why an active attacker, who has the ability to control the network, can modify all messages exchanged between two parties, leading to an MITM attack. However, in this paper, we illustrate a symbolic approach for reasoning about KEMs rather than focusing on reporting about this kind of attack. Because those three KEMs are relatively new, they need to be deeply analyzed to gain a confident security guarantee. Our ultimate goal is to conduct security analysis/verification of post-quantum cryptographic protocols, such as the post-quantum transport layer security (TLS) protocol [13]. Such protocols use post-quantum cryptographic primitives, such as KEMs analyzed in this paper, and then formally specifying such primitives is necessary to analyze the protocol security later on. What has been reported in this paper can be regarded as the first step toward the goal.

Authenticated key exchange (AKE) refers to the class of key exchange protocols in which authentication to participants is included in order to avoid MITM attacks. AKE has been extensively studied over the years, resulting in many protocols have been proposed, such as NAXOS [14]. The authentication solution in those protocols typically bases on a pair of ephemeral (like public key *pk* outputting by the algorithm `KeyGen`) and static (or long-term) keys. The static key provides authentication, while the ephemeral key provide (forward) secrecy. There also exist plenty of proposals for post-quantum AKE, such as by Zhou and Lv [15] and Ding et al. [16], which base their security on LWE.

8. Related Work

To the best of our knowledge, Jacomme et al. [17] and Hülsing et al. [18] are the only two case studies on analyzing post-quantum cryptographic protocols in the symbolic model. The former has presented a formal analysis of the EDHOC (Ephemeral Diffie Hellman Over COSE) protocol [19], a variant of the DH protocol designed by IETF's Lightweight AKE Working Group to be used in IoT devices. The original protocol uses the DH key exchange, which is not post-quantum secure because the discrete logarithm problem will be no longer hard with large-scale quantum computers. The protocol then was made post-quantum secure by replacing the DH with a post-quantum KEM. This KEM based version is also covered in their analysis. An interesting point in this work is that they used SAPIC$^+$ [20] protocol verification platform so that their formal specification written in pi-calculus can be exported into some other security analyzer tools including ProVerif [21] and Tamarin [22]. As we mentioned before, security analysis/verification of post-quantum cryptographic protocols is also our plan for the next step.

WireGuard [23] is a VPN protocol focusing on simplicity, fast speed, and high performance. Facing with the quantum attack threat, its quantum-resistant version has been proposed, namely post-quantum WireGuard (PQ-WireGuard) [18]. In that work, the authors have verified that PQ-WireGuard enjoys some desired security properties with the presence of large-scale quantum computers by using Tamarin [22], a well-known formal method tool for the symbolic analysis of cryptographic protocols. To do so, the security properties are formalized as Tamarin lemmas, and some auxiliary lemmas are introduced, where lemma conjecture is known as a creative and intellectual task in the formal verification. Additionally, the paper has also presented a computational security proof, which gave stronger security guarantees than the symbolic proof since probability and complexity are taken into account and fewer idealizing assumptions are made. However, more security properties are verified in the symbolic verification, and more importantly, the symbolic verification is computer-verified.

The most well-known symbolic cryptographic protocol analysis tools can be mentioned are ProVerif [21], Maude-NPA [24], Tamarin [22], and Scyther [25]. Based on the applied pi-calculus [26], ProVerif [21] can automatically verify security properties of a given cryptographic protocol. The verification can be achieved with the presence of a Dolev and Yao [8] intruder under an unbounded number of protocol executions. Using the applied pi-calculus, human users are supposed to model the cryptographic protocol, and then ProVerif translates it to a set of Horn clauses. This Horn clause representation makes some abstractions, which is the cost for the support of an unbounded number of sessions. Given a security property that we want to prove, the tool reduces the problem of finding an attack against the property to the derivability of a fact on the Horn clauses representing the protocol execution. If the fact is not derivable from the clauses, the property is proved. On the other hand, if the fact is derivable from the clauses, there may be an attack violating the property under analysis, but it may also be a "false attack," i.e., the found derivation actually does not correspond to a real attack.

Maude-NPA [24] is a powerful formal verification tool for analyzing the cryptographic protocols implemented in Maude. The strand space model [27] is used to model the protocol execution and the capabilities of the Dolev and Yao [8] intruder. For the analysis, the tool uses a backward narrowing reachability analysis modulo an equational theory, where narrowing is a generalization of term rewriting. Human users are supposed to specify the security property under verification as a Maude-NPA attack pattern (state) violating the property. Then, Maude-NPA performs the backward reachability analysis from that insecure pattern to check whether it can be reachable from an initial state. If that is the case, the attack is possible, namely, the property is violated; otherwise, the property is proven. The key feature

of Maude-NPA is that it supports many equational theories. But it may lead to the case a bigger state space is generated, making a long time for the backward reachability analysis to terminate, and so it requires some techniques to prune the search space. A key technique to do so is by generating formal grammars representing terms (states information) unreachable from initial states [28].

Tamarin [22], a successor of Scyther [25], is a cryptographic protocol analysis based on multiset rewriting [29]. A Tamarin specification consists of a set of *rules*, defining how states change from one to the next, modeling how a protocol is to be executed, how trustworthy parties are to behave, and what the Dolev and Yao [8] intruder is capable of. Each state is represented by an AC-collection of *facts*, and so a Tamarin specification can be regarded as a state machine. Roughly speaking, facts and rules correspond to observable components and Maude rewrite rules, respectively, as we described in this paper. A security property is modeled as a trace property, and then constraint solving is used to perform an exhaustive and symbolic search for executions with the trace until a satisfying one is found or no more rewrite rules can be applied. In addition to the *automated mode*, the tool also provides the *interactive mode*. When the tool does not terminate in the *automated mode*, human users are allowed to provide some additional lemmas to complete the proof. In the field of formal verification, conjecturing suitable lemmas as we all know is an intellectual task.

In the case study by Yadav et al. [30], the authors explored NTRU key exchange [31], a lattice-based public key exchange protocol, and found that it is exposed to an MITM attack. This MITM attack is similar to the attack reported in this paper, namely the attack caused by the lack of authentication. However, not like us, they used neither any tool nor formal specification language as we do.

The computational approach is widely used by cryptographers to prove the security of cryptosystems. An attacking game is typically used to define the security of a cryptographic primitive or protocol. The participants include an *adversary* (attacker) and a benign entity, a so-called where the *adversary* is an arbitrary probabilistic polynomial-time Turing machine. machine. The security proof can then be seen as a mathematical reduction that ensures the adversary is unable to gain an advantage over the challenger unless the adversary can solve some presumed computationally hard problem. When a proof becomes too complicated, the sequence of games technique may be employed. It is possible to mechanize security proofs in the computational approach to some extent with some supporting tools, such as CryptoVerif [32] and EasyCrypt [33, 34]. With the employment of CryptoVerif, Blanchet [35] have verified the security of the password-based key exchange protocol OEKE [36], a variant of the encrypted key exchange (EKE) protocol [37]. The mechanization technique in general is still not mature to apply to a wide range of protocols.

9. Conclusion and Future Work

We have presented the formal specifications and model checkings of the three lattice-based KEMs in Maude. We have first made a base specification where the common parts in the designs of the three KEMs are modeled. Then, the specification of each KEM has been completed by extending that base specification to model the mechanism execution, and thus, it makes less our effort in doing formal specification of the three KEMs. Afterward, we conducted invariant model checkings in Maude by using the `search` command, finding an MITM attack for each of the three KEMs. The attack occurs basically because a KEM alone is not equipped with any authentication solution, making it possible for an active attacker in the middle of connections with the network control ability to modify all communication between two honest parties.

Quantum-resistant versions of some cryptographic protocols have been proposed, such as post-quantum SSH [38] and post-quantum TLS protocol [13]. They share the same key idea, that is a classical key exchange algorithm (e.g., DH and Elliptic Curve DH) is either concurrently used with or completely replaced with another post-quantum KEM (e.g., Kyber and Saber). In the former case, the reason why a post-quantum KEM is required is clear, but why a classical key exchange algorithm is still needed. One reason is that such the KEM is not received enough confidence from the security point of view because it may not be studied/analyzed deeply. Therefore, deep security analysis of KEMs, other post-quantum primitives and protocols is an important challenge to guarantee their reliability. As a piece of our future work, we are going to conduct formal verification of the above-mentioned post-quantum protocols, i.e., post-quantum SSH and post-quantum TLS. As another piece of our future work, we are also interested in analyzing some AKE protocols, the AKE protocols, by using Maude or some other formal method tools.

Data Availability

All of the Maude specifications and checking commands reported in the paper are available at https://github.com/duongtd23/lattice-based-kems-mc.

Disclosure

A part of this study was accepted as a work-in-progress paper by the 14th International Workshop on Rewriting Logic and its Applications (WRLA 2022), however, that paper was not published in the workshop's proceedings.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

Tran and Ogata have been supported by JST SICORP (Grant Number JPMJSC20C2), Japan. Escobar has been partially supported by the grant PID2021-122830OB-C42 funded by MCIN/AEI/10.13039/501100011033 and ERDF A way of making Europe, by the grant CIPROM/2022/6 funded by Generalitat Valenciana, and by the grant PCI2020-120708-2 funded by MICIN/AEI/10.13039/501100011033 and by the

European Union NextGenerationEU/PRTR. Akleylek has been partially supported by TUBITAK under grant number 121R006. Otmani has been supported by the FAVPQC project funded by CNRS and by the grant ANR-22-PETQ-0008 PQ-TLS funded by Agence Nationale de la Recherche (ANR) within France 2030 program.

References

[1] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, IEEE, Santa Fe, NM, USA, November 1994.

[2] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *STOC '96: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pp. 212–219, Association for Computing Machinery, New York, NY, USA, July 1996.

[3] B. Blanchet, "Security protocol verification: symbolic and computational models," in *Principles of Security and Trust. POST 2012. Lecture Notes in Computer Science*, vol. 7215, pp. 3–29, Springer, Berlin, Heidelberg, 2012.

[4] J. Bos, L. Ducas, E. Kiltz et al., "CRYSTALS - kyber: a CCA-secure module-lattice-based KEM," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 353–367, IEEE, London, UK, April 2018.

[5] 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 *Progress in Cryptology - AFRICACRYPT. 2018 - International Conference on Cryptology in Africa, Marrakesh*, A. Joux, A. Nitaj, and T. Rachidi, Eds., pp. 282–305, vol. 10831 of *Lecture Notes in Computer Science*, Springer, Cham, Morocco, May 2018.

[6] S. Akleylek and K. Seyhan, "Module learning with rounding based key agreement scheme with modified reconciliation," *Computer Standards & Interfaces*, vol. 79, Article ID 103549, 2022.

[7] F. Durán, S. Eker, S. Escobar et al., "Programming and symbolic computation in maude," *Journal of Logical and Algebraic Methods in Programming*, vol. 110, Article ID 100497, 2020.

[8] 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.

[9] 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 *Proceedings of the International Workshop on Formal Analysis and Verification of Post-Quantum Cryptographic Protocols co-located with the 23rd International Conference on Formal Engineering Methods (ICFEM 2022)*, S. Akleylek, S. Escobar, K. Ogata, and A. Otmani, Eds., pp. 16–32, vol. 3280 of *CEUR Workshop Proceedings*, Madrid, Spain, October 2022.

[10] D. D. Tran, K. Ogata, S. Escobar, S. Akleylek, and A. Otmani, "Formal specification and model checking of saber lattice-based key encapsulation mechanism in maude," in *The 34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022, KSIR Virtual Conference Center, USA*, pp. 382–387, KSI Research Inc, July 2022.

[11] R. Avanzi, J. Bos, L. Ducas et al., "CRYSTALS-Kyber: algorithm specifications and supporting documentation (version 3.02)," 2021, https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[12] A. Basso, J. M. B. Mera, J.-P. D'Anvers et al., "SABER: Mod-LWR based KEM (round 3 submission)," 2017, https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf.

[13] M. Campagna and E. Crockett, "Hybrid post-quantum key encapsulation methods (PQ KEM) for transport layer security 1.2 (TLS)," 2021, RFC Editor https://datatracker.ietf.org/doc/html/draft-campagna-tls-bike-sike-hybrid.

[14] B. LaMacchia, K. Lauter, and A. Mityagin, "Stronger security of authenticated key exchange," in *Provable Security, International Conference on Provable Security, ProvSec 2007*, W. Susilo, J. K. Liu, and Y. Mu, Eds., vol. 4784 of *Lecture Notes in Computer Science*, pp. 1–16, Springer, Berlin, Heidelberg, Wollongong, Australia, 2007.

[15] L. Zhou and F. Lv, "A simple provably secure AKE from the LWE problem," *Mathematical Problems in Engineering*, vol. 2017, Article ID 1740572, 16 pages, 2017.

[16] J. Ding, P. Branco, and K. Schmitt, "Key exchange and authenticated key exchange with reusable keys based on RLWE assumption," *Cryptology ePrint Archive*, Article ID 665, 2019.

[17] C. Jacomme, E. Klein, S. Kremer, and M. Racouchot, "A comprehensive, formal and automated analysis of the EDHOC protocol," in *32nd USENIX Security Symposium (USENIX Security 23*, pp. 5881–5898, USENIX Association, Anaheim, CA, 2023.

[18] 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)*, pp. 304–321, IEEE, San Francisco, CA, USA, May 2021.

[19] G. Selander, J. P. Mattsson, and F. Palombini, "Ephemeral Diffie–Hellman over COSE (EDHOC)," 2022, Internet Engineering Task Force. draft-ietf-lake-edhoc-17. work in Progress, https://datatracker.ietf.org/doc/draft-ietf-lake-edhoc/17/.

[20] V. Cheval, C. Jacomme, S. Kremer, and R. Künnemann, "SAPIC+: protocol verifiers of the world, unite!" in *31st USENIX Security Symposium (USENIX Security 22*, pp. 3935–3952, USENIX Association, Boston, MA, USA, 2022.

[21] B. Blanchet, V. Cheval, and V. Cortier, "ProVerif with lemmas, induction, fast subsumption, and much more," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 69–86, IEEE, San Francisco, CA, USA, May 2022.

[22] D. Basin, C. Cremers, J. Dreier, and R. Sasse, "Symbolically analyzing security protocols using tamarin," *ACM SIGLOG News*, vol. 4, no. 4, pp. 19–30, 2017.

[23] J. A. Donenfeld, "WireGuard: next generation kernel network tunnel," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2017*, San Diego, CA, USA, 2017.

[24] S. Escobar, C. Meadows, and J. Meseguer, "Maude-NPA: cryptographic protocol analysis modulo equational properties," in *Foundations of Security Analysis and Design V*, A. Aldini, G. Barthe, and R. Gorrieri, Eds., pp. 1–50, Springer, Berlin, Heidelberg, Berlin, Heidelberg, 2009.

[25] C. J. F. Cremers, *Scyther - semantics and verification of security protocols*, Eindhoven University of Technology, Ph.D. thesis, 2006.

[26] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends® in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, 2016.

[27] F. J. T. Thayer, J. C. Herzog, and J. D. Guttman, "Strand spaces: why is a security protocol correct?" in *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat.*

No.98CB36186), pp. 160–171, IEEE, Oakland, CA, USA, May 1998.

[28] S. Escobar, C. Meadows, J. Meseguer, and S. Santiago, "State space reduction in the Maude-NRL protocol analyzer," *Information and Computation*, vol. 238, pp. 157–186, 2014.

[29] J. C. Mitchell, "Multiset rewriting and security protocol analysis," in *Rewriting Techniques and Applications*, S. Tison, Ed., vol. 2378, pp. 19–22, Springer, Berlin, Heidelberg, Copenhagen, Denmark, 2002.

[30] V. K. Yadav, S. Venkatesan, and S. Verma, "Man in the middle attack on NTRU key exchange," in *Communication, Networks and Computing*, S. Verma, R. S. Tomar, B. K. Chaurasia, V. Singh, and J. Abawajy, Eds., pp. 251–261, Springer Singapore, Singapore, 2019.

[31] X. Lei and X. Liao, "NTRU-KE: a lattice-based public key exchange protocol," *Cryptology ePrint Archive*, 2013, http://eprint.iacr.org/2013/718, Article ID 718.

[32] B. Blanchet, "Mechanizing game-based proofs of security protocols," in *Software Safety and Security - Tools for Analysis and Verification. vol. 33 of NATO Science for Peace and Security Series - D: Information and Communication Security*, T. Nipkow, O. Grumberg, and B. Hauptmann, Eds., pp. 1–25, IOS Press, 2012.

[33] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *Advances in Cryptology – CRYPTO 2011*, P. Rogaway, Ed., vol. 6841, pp. 71–90, Springer, Berlin, Heidelberg, Santa Barbara, CA, USA, 2011.

[34] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub, "Easycrypt: a tutorial," in *Foundations of Security Analysis and Design VII*, A. Aldini, J. López, and F. Martinelli, Eds., vol. 8604, pp. 146–166, Springer, Cham, 2013.

[35] B. Blanchet, "Automatically verified mechanized proof of one-encryption key exchange," in *2012 IEEE 25th Computer Security Foundations Symposium*, pp. 325–339, IEEE, Cambridge, MA, USA, June 2012.

[36] E. Bresson, O. Chevassut, and D. Pointcheval, "Security proofs for an efficient password-based key exchange," in *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pp. 241–250, Association for Computing Machinery, New York, NY, USA, October 2003.

[37] S. M. Bellovin and M. Merritt, "Encrypted key exchange: password-based protocols secure against dictionary attacks," in *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pp. 72–84, IEEE, Oakland, CA, USA, May 1992.

[38] P. Kampanakis, D. Stebila, and T. Hansen, "Post-quantum hybrid key exchange in SSH," 2023, Internet Engineering Task Force, draft-kampanakis-curdle-ssh-pq-ke-01. work in Progress, https://datatracker.ietf.org/doc/draft-kampanakis-curdle-ssh-pq-ke/01/.