Formal specification and model checking of lattice-based key encapsulation mechanisms in Maude*

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

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

Abstract

Advances in quantum computing have shown a serious challenge for widely used current cryptographic techniques because a sufficient large-scale quantum computer can efficiently solve hard mathematical problems on which the current public-key cryptography is relying. That is the reason why recently many researchers and industrial companies have spent lots of effort on constructing post-quantum cryptosystems, which are resistant to quantum attackers. Large numbers of post-quantum key encapsulation mechanisms (KEMs) have been proposed to provide secure key establishment - one of the most important building blocks in asymmetric cryptography. This paper presents a formal security analysis of three lattice-based KEMs: Kyber, Saber, and SK-MLWR. We first formally specify each of them in Maude, a rewriting logic-based specification and programming language equipped with many functionalities, such as a reachability analyzer (or the search command) that can be used as an invariant model checker, and then conduct invariant model checking with the Maude search command, finding an attack.

Keywords

key encapsulation mechanism, Maude, post-quantum, model checking

1. Introduction

In recent years, advanced research in the field of quantum computing and quantum information theory has brought a credible threat to cryptosystems currently in use. The most popular asymmetric (or public-key) primitives used today will become insecure against sufficiently

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

*D. D. Tran and K. Ogata have been supported by JST SICORP Grant Number JPMJSC20C2, Japan.
S. Akleylek has been partially supported by TUBITAK under Grant No.121R006.
S. Escobar has been partially supported by the grant RTI2018-094403-B-C32 funded by MCIN/AEI/10.13039/501100011033 and ERDF A way of making Europe, by the grant PROMETEO/2019/098 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.

*Corresponding author.
duongtd@jaist.ac.jp (D. D. Tran); ogata@jaist.ac.jp (K. Ogata); sescobar@upv.es (S. Escobar); sedat.akleylek@bil.omu.edu.tr (S. Akleylek); ayoub.otmani@univ-rouen.fr (A. Otmani)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3280/paper2.pdf

strong quantum computers because they can be efficiently broken by Shor's algorithm [1]. The security of these primitives relies on one of the following three hard mathematical problems: the integer factorization problem, the discrete logarithm problem, and the elliptic-curve discrete logarithm problem. All of these problems are hard under conventional computers, but they can be easily solved on a sufficiently powerful quantum computer running Shor's algorithm. On the other hand, symmetric primitives are considered secure against quantum attackers. Although Grover's algorithm [2], one of the most well-known quantum algorithms, can reduce the complexity to break symmetric primitives, doubling the key size can efficiently avoid these attacks. For example, we can say that AES-256 would be as hard to break by a quantum computer as AES-128 is by a classical computer.

As a response to the quantum attack threat, there is extensive research to find new schemes which are secure even in the presence of quantum adversaries. In the past few years, many post-quantum asymmetric primitives have been proposed as replacements for those traditional ones currently in use. The National Institute of Standards and Technology of USA (NIST) also started the Post-Quantum Cryptography Project in 2017, calling for proposals of post-quantum cryptographic protocols that are secure against both conventional and quantum computers[1]. There were 82 submissions to this standardization project, implying the importance of this problem. Among these submissions, a large number of proposals are for post-quantum key encapsulation mechanisms (KEMs), which aim to securely establish a symmetric key between two parties. This is understandable because key exchange algorithms are considered the most important building block of asymmetric cryptography.

Security analysis of cryptographic primitives and/or protocols can be fundamentally divided into two approaches: computational security and symbolic security. Proof in the computational model requires a definition of secure cryptographic construction (primitive, protocol), and some assumptions about the computationally infeasible problem. The proof can be regarded as a mathematical reduction to the situation where the only chance to violate the security of such a construction is to solve the infeasible problem. The authors of the three KEMs considered in this paper have already presented their security proofs in the computational model. However, such proofs are often not easy to understand for non-experts in cryptography. On the other hand, symbolic analysis is easier to understand, computer-verified and suitable for automation. Our approach presented in this paper belongs to the latter. Note that our approach can be applied to not only the three KEMs but also other KEMs and other kinds of primitives as well.

We formally specify and model check three KEMs: Kyber [3] (precisely CRYSTALS-Kyber), Saber [4], and SK-MLWR [5]. Because of space limitation, we choose Kyber as the only KEM to illustrate in this paper. The specifications of the other KEMs can be found at https://github.com/duongtd23/kems-mc. Kyber is a KEM whose security is based on the hardness of solving the learning-with-errors (LWE) problem, while the security of Saber and SK-MLWR relies on the hardness of the Module Learning With Rounding (MLWR) problem. All of them belong to the lattice-based cryptography. We use Maude [6], a programming/specification language based on rewriting logic, to formally specify the Dolev-Yao generic intruder [7] as well as these KEMs. By employing the Maude search command, a Man-In-The-Middle (MITM) attack is found for each KEM. Although this kind of attack is not a novel attack for KEMs, the formal specifications

[1] https://csrc.nist.gov/projects/post-quantum-cryptography

in Maude and the model checking experiments are worth reporting. Our ultimate goal is to come up with a new security analysis/verification technique for post-quantum cryptographic protocols, which use post-quantum cryptographic primitives, such as the three KEMs reported in this paper. Formally specifying such primitives is necessary for analyzing the security later on. What is described in the paper is our initial step toward the goal.

Related work. In 2012, Blanchet [8] has surveyed various approaches to security protocol verification in both symbolic model and computational model. In the symbolic model, there is a large number of tools existing for verifying cryptosystems, such as ProVerif [9], Maude-NPA [10], Tamarin [11], and Scyther [12]. The symbolic protocol verifier ProVerif, which was developed by Blanchet, can automatically prove security properties of cryptographic protocol specifications. ProVerif is based on an abstract representation of the protocol by a set of Horn clauses, and it determines whether the desired security properties hold by resolution on these clauses. The practicability of ProVerif has been demonstrated through case studies, such as [13, 14]. ProVerif can handle an unbounded number of sessions (executions) of protocols, but termination is not guaranteed in general because the resolution algorithm may not terminate. To mitigate this challenge, Escobar et al. [15] proposed some techniques to reduce the size of the search space in Maude-NPA, such as generating formal grammars representing terms (states information) unreachable from initial states and using super lazy intruder to delay the generation of substitution instances as much as possible. Even though, the termination of the tool is not always guaranteed. Among many case studies that demonstrated the capabilities of Maude-NPA, [16] presented one case study with Diffie-Hellman key agreement protocol.

Scyther [12] is another tool for symbolic security verification of cryptographic protocols. Like ProVerif, Scyther also supports an unbounded number of sessions, but it supports only a fixed set of cryptographic primitives (symmetric and asymmetric encryption and signatures) and does not allow for user-specified equational theories. Its successor, namely Tamarin [11] prover, does support equational theories. Moreover, Tamarin provides two ways of constructing proofs: fully automated mode and interactive mode. The tool may not terminate in the fully automated mode. In the interactive mode, the tool allows users to provide lemmas that must be proved. Several case studies on security analysis of cryptographic primitives and protocols with Tamarin can be found in [17, 18].

Yadav et al. [19] explored NTRU key exchange, a lattice-based public key exchange protocol, and found that it is exposed to an MITM attack. The attack was found in the same manner as what we present in this paper. However, they used neither any tool nor formal specification language as we do.

In the computational security approach, game-based model is known as a standard model for proving security. Security for cryptographic primitives or protocols is defined as an attack game played between an *adversary* and some benign entity, which is called the *challenger*. The main idea of the game-based security model is simulation of interaction among these two parties. Eventually, the security proof typically leads to a proof that any supposed adversary can get an advantage over the challenger if and only if he/she is able to solve some computationally infeasible problem (e.g., discrete logarithm, integer factorization). When a proof becomes too complicated, the proof normally employs the sequence of games technique [20]. CryptoVerif [21] is a tool for mechanizing such proof. It can generate proofs by sequences of games automatically

or with little user interaction. Alwen et al. [22] have employed CryptoVerif to analyze the security of the Hybrid Public Key Encryption (HPKE), which is a candidate for a new public key encryption standard.

Roadmap. The remaining of this paper is organized as follows: Section 2 gives some preliminaries, such as KEM and state machine. Section 3 describes Kyber KEM, briefly explains the underlying learning with error problem. The specification of Kyber in Maude is presented in Section 4. The model checking result and the attack found are presented in Section 5. Finally, Section 6 summarizes the paper.

2. Preliminaries

2.1. Key encapsulation mechanism

A key encapsulation mechanism is a tuple of algorithms (`KeyGen`, `Encaps`, `Decaps`) along with a finite keyspace $\mathcal{K}$:

- `KeyGen`() $\rightarrow (pk, sk)$: A probabilistic *key generation* algorithm that outputs a public key pk and a secret key sk.
- `Encaps`(pk) $\rightarrow (c, k)$: A probabilistic *encapsulation* algorithm that takes as input a public key pk, and outputs a ciphertext (or encapsulation) c and a key $k \in \mathcal{K}$.
- `Decaps`(c, sk) $\rightarrow k$: A (usually deterministic) *decapsulation* algorithm that takes as input a ciphertext c and a secret key sk, and outputs a key $k \in \mathcal{K}$.

2.2. State machine and Maude

A state machine $\mathcal{M} \triangleq \langle \mathcal{S}, \mathcal{I}, \mathcal{T} \rangle$ consists of a set $\mathcal{S}$ of states, a set $\mathcal{I} \subseteq \mathcal{S}$ of initial states and a binary relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ over states. The set $\mathcal{R}$ of *reachable states* with respect to $\mathcal{M}$ is inductively defined as follows: (1) for each $s \in \mathcal{I}$, $s \in \mathcal{R}$ and (2) for each $(s, s') \in T$, if $s \in \mathcal{R}$, then $s' \in \mathcal{R}$. A state predicate p is an invariant property with respect to $\mathcal{M}$ if and only if $p(s)$ holds for all $s \in \mathcal{R}$.

In this paper, to express a state of $\mathcal{S}$, we use a braced associative-commutative collection of name-value pairs. Associative-commutative collections are called soups, and name-value pairs are called observable components. That is, a state is expressed as a soup of observable components. The juxtaposition operator is used as the constructor of soups. Let $oc1, oc2, oc3$ be observable components, and then $oc1\ oc2\ oc3$ is the soup of those three observable components. A state is of the form $\{oc1\ oc2\ oc3\}$. There are multiple possible ways to specify state transitions. In this paper, we use Maude [6], a programming and specification language based on rewriting logic, to specify them as rewrite rules. Maude makes it possible to specify complex systems flexibly and it is also equipped with model checking facilities (a reachability analyzer and an LTL model checker). A rewrite rule starts with the keyword **`rl`**, followed by a label enclosed with square brackets and a colon, two patterns connected with =>, and ends with a full stop. A conditional one starts with the keyword **`crl`** and has a condition following the keyword **`if`** before a full stop. The following is a form of a conditional rewrite rule:

```
crl [lb] : l => r if ... /\ c_i /\ ... .
```

where lb is a label and 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, $\sigma(l)$ can be replaced with $\sigma(r)$.

Maude provides the **search** command that can find a state reachable from t such that the state matches p and satisfies the condition(s) c:

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

where MOD is the name of the module specifying the state machine, and n and m are optional arguments denoting a bound on the number of desired solutions and the maximum depth of the search. n typically is 1 and t typically represents an initial state of the state machine.

3. Kyber Key Encapsulation Mechanism

3.1. Notations

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

The ring of integers modulo q is denoted by $\mathbb{Z}_q$. We denote by R the polynomial ring $\mathbb{Z}[X]/(X^n + 1)$ and by R_q the quotient polynomial ring $\mathbb{Z}_q[X]/(X^n + 1)$. Thus polynomials in R_q are of n coefficients where each coefficient is in $[0, q)$. In Kyber, the values of n and q are always fixed to $n = 256$ and $q = 7681$ in all levels of security. Regular font letters denote elements in R or R_q (which includes elements in $\mathbb{Z}$ and $\mathbb{Z}_q$) and bold lower-case letters represent vectors with coefficients in R or R_q. By default, all vectors are column vectors. Bold upper-case letters are matrices. For a vector $\mathbf{v}$ (or matrix $\mathbf{A}$), we denote by $\mathbf{v}^T$ (or $\mathbf{A}^T$) its transpose. For a vector $\mathbf{v}$ we write $\mathbf{v}[i]$ to denote its i-th entry (with indexing starting at zero); for a matrix $\mathbf{A}$ we write $\mathbf{A}[i][j]$ to denote the entry in row i and column j (again, with indexing starting at zero).

Let H and G be two hash functions, where $\mathrm{H} : \mathcal{B}^* \to \mathcal{B}^{32}$ and $\mathrm{G} : \mathcal{B}^* \to \mathcal{B}^{32} \times \mathcal{B}^{32}$. Let KDF denote the key derivation function, where $\mathrm{KDF} : \mathcal{B}^* \to \mathcal{B}^{32}$. Let $x \in \mathbb{Q}$, where $\mathbb{Q}$ denotes the rational numbers set, then $\lfloor x \rceil$ denotes rounding of x to the closest integer. Let $x \in \mathbb{Z}_q$ and $d < \lfloor \log_2 q \rceil$, functions Compress and Decompress are defined by the following equations:

$$\mathrm{Compress}_q(x, d) = \lfloor (2^d/q) \cdot x \rceil \bmod 2^d, \tag{1}$$

$$\mathrm{Decompress}_q(x, d) = \lfloor (q/2^d) \cdot x \rceil \tag{2}$$

Let $x' = \mathrm{Decompress}_q(\mathrm{Compress}_q(x, d), d)$, then we have:

$$|x' - x \bmod q| \leq \lfloor \frac{q}{2^{d+1}} \rceil \tag{3}$$

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

KEM.KeyGen()
$z \leftarrow \mathcal{B}^{32}$
$(pk, sk') = \mathsf{CPAPKE.KeyGen}()$
$sk = (sk'||pk||\mathrm{H}(pk)||z)$
return (pk, sk)

$\xrightarrow{pk}$

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

$\xleftarrow{c}$

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

Figure 1: Kyber.KEM

3.2. Kyber

Fig. 1 describes the three algorithms (KeyGen, Encaps, Decaps) of Kyber KEM. It employs the three algorithms (KeyGen, Enc, Dec) of Kyber.CPAPKE, which are shown in Fig. 2. Let us suppose that there are two parties Alice and Bob. The interactions depicted in Fig. 1 are as follows. Alice performs the KEM.KeyGen step, starting by choosing a random seed d, and hashing it to get the pair (ρ, σ). From σ, she generates vector $\mathbf{s}$ serving as the secret key sk and vector $\mathbf{e}$ acting as an error component. $\mathbf{t}$ is then computed from $\mathbf{s}$, $\mathbf{e}$, and matrix $\mathbf{A}$, which is generated from ρ. The pair of $\mathbf{t}$ and ρ serves as the public key pk, which is sent to Bob. Upon receiving pk, Bob executes the KEM.Enc step (i.e., Encaps step) in Fig. 1. He randomly chooses an m_0, hashes it, and passes the outputs to the CPAPKE.Enc procedure (depicted in Fig. 2). The obtained ciphertext c, which is a pair of c_1 and c_2, is sent back to Alice. Upon receiving c, Alice performs the KEM.Dec step (i.e., Decaps step). She computes c' by employing the CPAPKE.Dec and CPAPKE.Enc procedures. With a very high probability c' is equal to c, implying that m' on Alice's side is equal to m on Bob's side with an overwhelming probability. After that, they can derive the same key K. Note that all multiplications and additions in the two figures are computed over $\mathbb{Z}_q[X]/(X^n + 1)$.

Note that in [3], the definition of Kyber employs the functions Encode and Decode. Function Encode serializes a polynomial or a vector of polynomials to a byte array, and function Decode is the inverse of Encode. Furthermore, to perform multiplications in R_q efficiently, the vectors and matrices are converted to NTT domain and vice versa, where NTT stands for number-theoretic transform. However, implementation or performance is out of the scope of the present paper; thus, for simplicity and ease of understanding, we omit those concepts. Consequently, the notation, e.g., $pk = (\mathbf{t}||\rho)$ is a misuse of notations because $\mathbf{t} \in R_q^k$ and then t is not a byte array. This notation is understood as pk is made of $\mathbf{t}$ and ρ.

The procedure to generate matrix $\mathbf{A}$, which is denoted by $\mathrm{generate}(\rho)$ in Fig. 2, taking as input a random seed ρ, is deterministic. Informally, if Alice and Bob share the same random seed ρ, then they can agreeingly derive the same matrix $\mathbf{A}$, whose coefficients of each entry are close to a uniformly random distribution. In contrast, the procedure to sample noise (or error)

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

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 $c = (c_1||c_2)$

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'

Figure 2: Kyber.CPAPKE

components (e.g., $\mathbf{e}$, $\mathbf{e_1}$, and e_2), namely $\mathrm{sampleCBD}$, is probabilistic. It takes as input a random seed (e.g., ρ and r) and returns a polynomial whose coefficients are close to a centered binomial distribution (to sample a vector, e.g., $\mathbf{s}$ and $\mathbf{e}$, the procedure is called k times). Informally, coefficients of an output of $\mathrm{sampleCBD}$ are mostly close to 0, and their absolute value is never greater than a specific small number (which is 5, or 4, or 3, depending on the level of security).

4. Formal specification of Kyber

4.1. Formalization of polynomials, vectors, and matrices

We first introduce sort `Poly` that represents polynomials as follows:

```
sort Poly . subsort Int < Poly .
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 `Int` is the sort of integers. The notation **`subsort`** `Int < Poly` indicates that any integer is also a polynomial. `p+`, `p*`, and `p-` denote the addition, multiplication, and subtraction, respectively, between two polynomials. `neg` denotes the negation of a polynomial. `assoc comm` indicates that `_p+_` and `_p*_` are declared to be associative and commutative. `prec 33` attached with `_p+_` and `_p-_` indicates that these operators have the same precedence 33, which is lower precedence than that of `_p*_` (i.e., `31`). Note that, we only consider polynomials in $\mathbb{Z}_q[X]/(X^n + 1)$ (or R_q), where $n = 256$ and $q = 7681$. Let `P1`, `P2`, and `P3` be variables of `Poly`. We declare some properties of the operators as follows:

```
eq P1 p+ 0 = P1 .                eq P1 p* 0 = 0 .        eq P1 p* 1 = P1 .
eq P1 p* (P2 p+ P3) = (P1 p* P2) p+ (P1 p* P3) .
```

```
eq P1 p- P2 = P1 p+ neg(P2) .     eq P1 p+ neg(P1) = 0 .
eq neg(neg(P1)) = P1 .            eq neg(P1 p+ P2) = neg(P1) p+ neg(P2) .
```

In a similar way, we introduce sorts `Vector` and `Matrix` representing polynomial vectors and matrices, respectively; operators `v+`, `dot`, and `m*` representing the addition & inner product of two polynomial vectors, and multiplication of a polynomial matrix and a vector, respectively. Let `V1`, `V2`, and `V3` be variables of `Vector`. The declarations of the three operators and the distributive property of vectors are specified as follows:

```
op _v+_ : Vector Vector -> Vector [assoc comm prec 33] .
op _dot_ : Vector Vector -> Poly [prec 31] .
op _m*_ : Matrix Vector -> Vector [prec 31] .
eq (V1 v+ V2) dot V3 = (V1 dot V3) p+ (V2 dot V3) .
eq V3 dot (V1 v+ V2) = (V3 dot V1) p+ (V3 dot V2) .
```

4.2. Formalization of honest parties

Two constructors for the two kinds of messages used in Kyber are as follows:

```
op msg1 : Prin Prin Prin Vector Poly MsgState -> Msg [ctor] .
op msg2 : Prin Prin Prin Vector Poly MsgState -> Msg [ctor] .
```

where `Prin` is the sort representing principals, and `Msg` is the sort denoting messages. `MsgState` is the sort representing message states, receiving one of the following three values: `sent` - the message was sent, `replied` - the message was sent and the receiver replied with another message, and `intercepted` - the message was intercepted by the intruder. The first, second, and third arguments of each operator are the actual creator, the seeming sender, and the receiver of the corresponding message. The first and last arguments are meta-information that is only available to the outside observer, while the remaining arguments can be seen by every principal. The fourth and fifth arguments of `msg1` carry the vector $\mathbf{t}$ and the random seed ρ, respectively, of the public key pk (pk is $(\mathbf{t}||\rho)$ as explained in Section 3.2). Similarly, the fourth and fifth arguments of `msg2` carry c_1 and c_2, respectively, of CPAPKE.

We model the network as a multiset of messages, in which the intruder can use as his/her storage. Consequently, the empty network (i.e., the empty multiset) means that no messages have been sent. The intruder can fully control the network, that is he/she can intercept any message, glean information from it, and fake a new message to any honest party. To formally specify Kyber in Maude, we use the following observable components:

- (`nw` : $msgs$) - $msgs$ is the soup of messages in the network;
- (`keys[p]` : $keys$) - $keys$ is a soup of the computed shared keys of principal `p`. Each entry of $keys$ is in form of `key(K,q)`, where `K` is the shared key and `q` is the principal whom `p` believes that he/she has communicated with;
- (`prins` : ps) - ps is the collection of all principals participating in the protocol;
- (`d[p]` : d_0) - d_0 is the random seed d (used in Fig. 2) of principal `p`;
- (`m[p]` : m_0) - m_0 is the random seed m_0 (shown in Fig. 1)) of principal `p`;

- (`rd-d` : $rdds$) - $rdds$ is a list of available values as the random seed d (we use list, but not set, to reduce the state space for searching). Each time when a principal makes a query for a random value of d, the top value in $rdds$ is removed and returned to the principal;
- (`rd-m` : $rdms$) - $rdms$ is a list of the available values as random seed m_0;
- (`glean-keys` : $gkeys$) - $gkeys$ is the soup of shared keys gleaned by the intruder;
- (`ds` : ds) - ds is the collection of the random seeds d used by the intruder. Note that every entry in ds is different from any random value used by honest parties;
- (`ms` : ms) - ms is the collection of random seeds m used by the intruder. Similarly to `ds`, every entry in ms is different from any random value used by honest parties.

Each state in $\mathcal{S}_{\text{Kyber}}$ is expressed as $\{obs\}$, where obs is a soup of those observable components. We suppose that there are two honest principals `alice` and `bob` together with a malicious one, namely `eve`, participating in Kyber KEM. The initial state `init` of $\mathcal{I}_{\text{Kyber}}$ is defined as follows:

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

With the honest parties, we specify three transitions: `keygen`, `encaps`, and `decaps`, which correspond to the three steps of the mechanism. Let `OCs` be a variable of observable component soups, `A`, `B`, and `C` be variables of principals (possibly intruder), and `PS` be a variable of principal collections. Let `D`, `M`, `M2`, `M'`, `Rho`, `RhoA`, `V`, `V'`, `CV`, and `P1` be variables of polynomials, and `PoL` be a variable of polynomial lists. Let `G` and `H` denote the hash functions G and H, respectively. Let `MS` be a variable of networks (i.e., message soups). The rewrite rule `keygen` is defined as follows:

```
crl [keygen] : {(rd-d: (D, PoL)) (d[A]: P1) (prins: (A B PS)) (nw: MS) OCs}
=> {(rd-d: PoL) (d[A]: D) (prins: (A B PS))
    (nw: (MS msg1(A,A,B, sample-A(1st(RhoSig)) m* sample-s(2nd(RhoSig))
     v+ sample-e(2nd(RhoSig)), 1st(RhoSig), sent))) OCs}
if RhoSig := G(D) .
```

where `RhoSig` is a variable denoting a pair of polynomials, `1st` and `2nd` are its projection operators. `sample-A`, `sample-e`, and `sample-s` represent the sampling procedures, outputting the matrix $\mathbf{A}$, the vectors $\mathbf{e}$, and $\mathbf{s}$, respectively. The rewrite rule says that when there exists a polynomial `D` in `rd-d`, `A` picks it as a random seed d, builds a message `msg1` exactly following the `KeyGen()` step of the mechanism, and sends it to `B`. `d[A]` is set to `D`, and `D` is removed from `rd-d`.

The rewrite rule `encap` is defined as follows:

```
crl [encap] : {(rd-m: (M, PoL)) (m[B]: P1) (keys[B]: KS)
   (nw: (msg1(C,A,B,T,Rho,sent) MS)) OCs}
=> {(rd-m: PoL) (m[B]: M) (keys[B]: (KS key(KDF(1st(Kr), H'(CU,CV)), A)))
    (nw: (msg1(C,A,B,T,Rho,replied) msg2(B,B,A, CU, CV, sent) MS)) OCs}
if M' := H(M) /\ Kr := G(pair(M', H'(T, Rho))) /\
   CU := enc-u(T, Rho, M', 2nd(Kr)) /\ CV := enc-v(T, Rho, M', 2nd(Kr)) .
```

where T and CU are variables of polynomial vectors, Kr is a variable denoting a pair of polynomials, and KS is a variable representing a soup of shared keys. Let Rseed be a variable of polynomials. Let sample-r, sample-e1, and sample-e2 represent the procedures sampling $\mathbf{r}$, $\mathbf{e_1}$, and e_2, respectively. Following the CPAPKE.Enc(pk, m, r) in Fig. 2, enc-u and enc-v are defined as follows:

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

where tp(A) denotes the transpose matrix of A and tpV(T) denotes the transpose vector of T. du and dv are constants of natural numbers, denoting du and dv, respectively. decompr(M,1) and compr(M,1) denote $\mathrm{Decompress}_q(\mathtt{M}, 1)$ and $\mathrm{Compress}_q(\mathtt{M}, 1)$, respectively. enc-u(T,Rho,M,Rseed) and enc-v(T,Rho,M,Rseed) compute c_1 and c_2, respectively, in Fig. 2 given as inputs T||Rho, M, Rseed. The rewrite rule encaps says that when there exists a message msg1 sent from A to B in the network, B builds a message msg2 exactly following the Encaps() step of Kyber, sends it back to A. B also computes the shared key with A, and the state of the message msg1 is updated to replied.

The rewrite rule decaps is defined as follows:

```
crl [decaps] : {(d[A]: D) (keys[A]: KS)
   (nw: (msg1(A,A,B,T,Rho,MsgStat) msg2(C,B,A, CU, CV, sent) MS)) OCs}
=> {(d[A]: D) (keys[A]: (KS key(KDF(1st(Kr2), H'(CU,CV)), B)))
    (nw: (msg1(A,A,B,T,Rho,MsgStat) msg2(C,B,A, CU, CV, replied) MS)) OCs}
if RhoSig := G(D) /\ Rho == 1st(RhoSig) /\
  T == sample-A(Rho) m* sample-s(2nd(RhoSig)) v+ sample-e(2nd(RhoSig)) /\
  U' := decompr(CU, du) /\ V' := decompr(CV, dv) /\
  M2 := compr(V' p- tpV(sample-s(2nd(RhoSig))) dot U', 1) /\
  Kr2 := G(pair(M2, H'(T, Rho))) /\
  enc-u(T,Rho,M2,2nd(Kr2)) == CU /\ enc-v(T,Rho,M2,2nd(Kr2)) == CV .
```

where MsgStat is a variable representing an arbitrary message state. The rewrite rule says that when A has sent a message msg1 to B and there exists a message msg2 replied from B to A in the network, A follows the Decaps() step of Kyber, computes the shared key with B. We only consider the overwhelming case, i.e., Alice successfully recovers m. We assume that the error tolerance gaps made by error components always be silent, making m' equals to m. This is done by the following equation:

```
ceq compr(E0 p+ decompr(M,1),1) = M if isSmall?(E0) .
```

where E0 is a variable of sort Poly, and isSmall?(E0) is a predicate, returning true if all coefficients of E0 are small in comparison with q. Sampling procedures for $\mathbf{s}$, $\mathbf{e}$, $\mathbf{r}$, $\mathbf{e_1}$, and e_2 return vectors or polynomials whose coefficients are small. These properties are specified by the following equations:

```
eq isSmall?(sample-s(P)) = true .      eq isSmall?(sample-e(P)) = true .
eq isSmall?(sample-e1(P)) = true .     eq isSmall?(sample-e2(P)) = true .
eq isSmall?(sample-r(P)) = true .
```

Using Eq. 3, we rewrite $\mathrm{Decompress}_q(\mathrm{Compress}_q(v, dv), dv)$ and $\mathrm{Decompress}_q(\mathrm{Compress}_q(\mathbf{u}, du), du)$ by $v + \epsilon_1$ and $\mathbf{u} + \boldsymbol{\epsilon}_2$, respectively, where all coefficients of ϵ_1 and $\boldsymbol{\epsilon}_2$ are small in comparison with those of v and $\mathbf{u}$. In the specification, we specify ϵ_1 as `epsilon1`(v), $\boldsymbol{\epsilon}_2$ 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 .
```

4.3. Formalization of intruders

We suppose that there is one intruder, namely `eve`, participating in the mechanism. When there exists a message `msg1` sent from `A` to `B` in the network, the intruder can intercept that message, fake a new message, and send it to the receiver. This behavior is specified by the following rewrite rule:

```
crl [keygen-eve] : {(ds: (D PoC1)) (nw: (msg1(A,A,B,TA,RhoA,sent) MS)) OCs}
=> {(ds: (D PoC1)) (nw: (msg1(A,A,B,TA,RhoA,intercepted)
   msg1(eve,A,B,sample-A(1st(RhoSig)) m* sample-s(2nd(RhoSig)) v+
   sample-e(2nd(RhoSig)),1st(RhoSig),sent) MS)) OCs}
if RhoSig := G(D) .
```

where `PoC1` and `PoC3` are variables representing arbitrary soups of polynomials. The intercepted message must have state `sent` at the beginning, which means that the message has not reached the receiver. `eve` then constructs a new faking message from an available value `D` for the random seed d. This kind of random value cannot be gleaned from the network, but `eve` can only construct it by randomly choosing a new value as the rewrite rule `build-ds` as follows:

```
rl [build-ds] : {(rd-d: (D, PoL)) (ds: PoC1) OCs}
=> {(rd-d: PoL) (ds: (PoC1 D)) OCs} .
```

Similarly, the only way in which `eve` can construct values for the random seed m is by randomly choosing a new value. This is specified by the following rewrite rule `build-ms`:

```
rl [build-ms] : {(rd-m: (M, PoL)) (ms: PoC3) OCs}
=> {(rd-m: PoL) (ms: (PoC3 M)) OCs} .
```

Two more rewrite rules are introduced as follows:

```
crl [encaps-eve] : {(ms: (M PoC3)) (glean-keys: KS)
  (nw: (msg1(A,A,B,TA,RhoA,intercepted) MS)) OCs}
=> {(ms: (M PoC3)) (glean-keys: (key(KDF(1st(Kr),H'(CU,CV)),A) KS))
    (nw: (msg1(A,A,B,TA,RhoA,intercepted) msg2(eve,B,A,CU,CV,sent) MS)) OCs}
if M' := H(M) /\ Kr := G(pair(M',H'(TA,RhoA))) /\
  CU := enc-u(TA,RhoA,M',2nd(Kr)) /\ CV := enc-v(TA,RhoA,M',2nd(Kr)) /\
  msg2(eve,B,A,CU,CV,sent) \in MS = false /\
  msg2(eve,B,A,CU,CV,replied) \in MS = false .

crl [decaps-eve] : {(ds: (D PoC1)) (glean-keys: KS)
  (nw: (msg1(eve,A,B,T,Rho,replied) msg2(B,B,A,CUB,CVB,sent) MS)) OCs}
```

```
=> {(ds: (D PoC1)) (glean-keys: (key(KDF(1st(Kr2), H'(CUB, CVB)), B) KS))
    (nw: (msg1(eve,A,B,T,Rho,replied) msg2(B,B,A,CUB,CVB,intercepted) MS)) OCs}
if RhoSig := G(D) /\ Rho == 1st(RhoSig) /\
  T == sample-A(Rho) m* sample-s(2nd(RhoSig)) v+ sample-e(2nd(RhoSig)) /\
  UB' := decompr(CUB, du) /\ VB' := decompr(CVB, dv) /\
  M2 := compr(VB' p- tpV(sample-s(2nd(RhoSig))) dot UB', 1) /\
  Kr2 := G(pair(M2, H'(T, Rho))) /\
  enc-u(T,Rho,M2,2nd(Kr2)) == CUB /\ enc-v(T,Rho,M2,2nd(Kr2)) == CVB .
```

`encaps-eve` says that when `eve` has intercepted a message `msg1` sent from `A` to `B`, `eve` fakes a new message `msg2`, sends it to `A`, and computes a shared secret key with `A`. `decaps-eve` says that when `eve` has faked a new message `msg1`, sent it to `B`, and `B` on his/her belief that the message truly comes from `A` has replied to `A` a message `msg2`, `eve` intercepts the message `msg2`, and computes a shared secret key with `B`.

5. Model checking and Man-In-The-Middle-Attack

We introduce the following search command:

```
search [1] in KYBER : init =>*
   {(keys[alice]: key(K1,bob)) (keys[bob]: key(K2,alice))
    (glean-keys: (key(K1,alice) key(K2,bob) KS)) OCs} .
```

where `K1` and `K2` are variables that denote arbitrary shared keys. `K1` may or may not be equal to `K2`. The command tries to find a state reachable from `init` such that: `alice` in her belief obtains the shared key `K1` with `bob`, `bob` in his belief obtains the shared key `K2` with `alice`, and `eve` owns both `K1` and `K2`. Maude found a counterexample, and this kind of vulnerability belongs to MITM attacks. Fig. 3 shows how this attack happens on Kyber, which is visualized from the path leading to the counterexample Maude returned. There are mainly five steps as follows:

Step 1. Alice wants to construct a shared key with Bob, she starts by performing KEM.`KeyGen()`, generating a public key pk and a secret key sk. She keeps sk, and sends pk to Bob.

Step 2. Eve intercepts the first message sent from Alice to Bob. She takes a random d_e, follows the KEM.`KeyGen()` step to generate a pair (pk_e, sk_e), and sends pk_e to Bob.

Step 3. Bob receives pk_e thinking it is from Alice. As a response, he takes a random m_0, performs KEM.`Enc(`pk_e`)`, and obtains a ciphertext c and a shared key K_b. He sends the ciphertext c back to Alice, and keeps the key K_b, which he believes that it is the shared key obtained by him and Alice.

Step 4. Eve intercepts the replied message which contains ciphertext c sent from Bob to Alice. She first performs KEM.`Dec(`c, sk_e`)` to obtain the shared key K_b. She then takes a random m_{e0}, performs KEM.`Enc(`pk`)`, and obtains a ciphertext c_e and a shared key K_a. She sends the ciphertext c_e back to Alice as a response for the first message.

Step 5. Alice receives the ciphertext c_e thinking it is from Bob. She performs KEM.`Dec(`c_e, sk`)` to obtain the shared key K_a. She believes that K_a is the shared key obtained by her and Bob.

The reachable state space in the experiment is finite. Indeed, if we try to run the following command: `search in KYBER : init =>* {OCs} .`, the number of returned solutions is finite,

Alice		Eve		Bob
$z \leftarrow \mathcal{B}^{32}$ $(pk, sk') = \mathsf{PKE.KeyGen}()$ $sk = (sk' \|\| pk \|\| \mathrm{H}(pk) \|\| z)$ $\mathbf{return}\ (pk, sk)$	$\xrightarrow{pk}$	$d_e \leftarrow \mathcal{B}^{32}$ $(\rho_e, \sigma_e) = \mathrm{G}(d_e)$ $R_q^{k\times k} \ni \mathbf{A}_e = \mathrm{generate}(\rho_e)$ $R_q^k \ni \mathbf{s}_e, \mathbf{e}_e \leftarrow \mathrm{sampleCBD}(\sigma_e)$ $\mathbf{t}_e = \mathbf{A}_e\mathbf{s}_e + \mathbf{e}_e$ $pk_e = (\mathbf{t}_e \|\| \rho_e)$ $sk_e = \mathbf{s}_e$ $\mathbf{return}\ (pk_e, sk_e)$		
			$\xrightarrow{pk_e}$	$m_0 \leftarrow \mathcal{B}^{32}$ $m = \mathrm{H}(m_0)$ $(\bar{K}, r) = \mathrm{G}(m \|\| \mathrm{H}(pk_e))$ $c = \mathsf{PKE.Enc}(pk_e, m, r)$ $K_b = \mathrm{KDF}(\bar{K}, \mathrm{H}(c))$ $\mathbf{return}\ (c, K_b)$
		$m'_b = \mathsf{PKE.Dec}(c, \mathbf{s}_e)$ $(\bar{K}'_b, r'_b) = \mathrm{G}(m'_b \|\| \mathrm{H}(pk_e))$ $c'_b = \mathsf{PKE.Enc}(pk_e, m'_e, r'_e)$ $\mathbf{if}\ c = c'_b\ \mathbf{then}$ $\mathbf{return}\ K_b = \mathrm{KDF}(\bar{K}'_b, \mathrm{H}(c))$	$\xleftarrow{c}$	
		$m_{e0} \leftarrow \mathcal{B}^{32}$ $m_e = \mathrm{H}(m_{e0})$ $(\bar{K}_e, r_e) = \mathrm{G}(m_e \|\| \mathrm{H}(pk))$ $c_e = \mathsf{PKE.Enc}(pk, m_e, r_e)$ $K_a = \mathrm{KDF}(\bar{K}_e, \mathrm{H}(c_e))$ $\mathbf{return}\ (c_e, K_a)$		
$m' = \mathsf{PKE.Dec}(c_e, \mathbf{s})$ $(\bar{K}', r') = \mathrm{G}(m' \|\| \mathrm{H}(pk))$ $c' = \mathsf{PKE.Enc}(pk, m', r')$ $\mathbf{if}\ c_e = c'\ \mathbf{then}$ $\mathbf{return}\ K_a = \mathrm{KDF}(\bar{K}', \mathrm{H}(c_e))$	$\xleftarrow{c_e}$			

Figure 3: A counterexample found by Maude (note that we use PKE as an abbreviation for CPAPKE to save space)

implying that the state space is finite. We give a brief explanation of this fact. Each state is denoted as a braced associative-commutative soup of the ten observable components as shown in Section 4. The key point is that the numbers of possible values that each observable component (i.e., a name-value pair) can receive is finite. Indeed, ps in (`prins` : ps) is always `(alice bob eve)` because there is no rewrite rule that changes it. $rdds$ and $rdms$ in (`rd-d` : $rdds$) and (`rd-m` : $rdms$) never consist of more than `(d1,d2)` and `(m1,m2)`, respectively, because there is no rewrite rule that inserts element(s) into them. d_0 and m_0 in (`d[p]` : d_0) and (`m[p]` : m_0) can only be in the sets {`d1`, `d2`} and {`m1`, `m2`}, respectively. Similarly, the numbers of possible values for ds and ms in (`ds` : ds) and (`ms` : ms) are finite. $msgs$ in (`nw` : $msgs$) consists of finite messages because (1) each of the two rewrite rules `keygen` and `encaps` adds a new message into the network, but simultaneously it also removes one element from $rdds$ and $rdms$ (note that $rdds$ and $rdms$ never consist of more than `(d1,d2)` and `(m1,m2)` as shown above); (2) the rewrite rule `keygen-eve` adds a new message `msg2` into the network, but simultaneously it also changes the status of an existing message `msg1` from `sent` to `intercepted`, thus, `keygen-eve`

can only be applied finitely many times; and (3) the rewrite rule `encaps-eve` only adds a new message `msg2` into the network if that message does not exist before (note that the other rewrite rules does not change the network or only update the status of messages). Similarly, $keys$ in (`keys[p]` : $keys$) consists of finite entries because: the rewrite rule `encaps` removes one element from $rdms$; and the rewrite rule `decaps` changes the status of an existing message `msg2` from `sent` to `replied`. In the same manner, we can show $gkeys$ in (`glean-keys` : $gkeys$) is finite. In summary, we can conclude that the state space in our experiment is finite. Consequently, with a search command to find a state satisfying some conditions, in finite time Maude will either find no solutions or will find a state satisfying the conditions.

Remark. Readers may argue that this kind of attack is not a novel attack since Kyber is not equipped with any feature for dealing with authentication. We agree on it. The paper instead illustrates one symbolic approach for reasoning about KEMs rather than focusing on this kind of attack. Our ultimate goal is to come up with a new security analysis/verification technique for post-quantum cryptographic protocols, such as post-quantum TLS. Such protocols use post-quantum cryptographic primitives, such as KEMs. Formally specifying such primitives is necessary to analyze the security. What is described in the paper is our initial step toward the goal.

Saber and SK-MLWR. In the same manner, we formalize the two other KEMs, i.e., Saber [4] and SK-MLWR [5], specify them in Maude, and run Maude search command trying to find the same kind of attack. The same MITM attacks are found by Maude. The reason is similar to the Kyber case study, that is because there is no authentication, and thus, the intruder can impersonate any party. We do not present these two case studies in this paper, but readers can find the formal specifications on the webpage presented in Section 1.

6. Conclusion

We have presented an approach to security analysis of some lattice-based KEMs in the symbolic model. We first used Maude as a specification language to formally specify the KEMs. After that, by employing Maude search command, an MITM attack was found for each KEM. The occurrence of the attack is basically because a KEM alone does not come with an authentication solution.

Researchers have proposed a post-quantum TLS protocol [23] that uses a hybrid key exchange method: a traditional key exchange algorithm together with a post-quantum KEM. The reason why a post-quantum KEM is required is clear. However, why do we still need to employ a traditional key exchange algorithm. One reason is that most post-quantum KEMs are not studied/analyzed deeply, and thus, nothing guarantees that there is not any potential flaw in them. Thus, deep security analysis of such KEMs in particular and other post-quantum cryptographic primitives/protocols is an important challenge to guarantee their reliability. One piece of our future work is to formally verify the security of the post-quantum TLS protocol against both classical and quantum computers.

References

[1] P. Shor, Algorithms for quantum computation: discrete logarithms and factoring, in: Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 124–134. doi:10.1109/SFCS.1994.365700.

[2] L. K. Grover, A fast quantum mechanical algorithm for database search, in: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, STOC '96, Association for Computing Machinery, New York, NY, USA, 1996, p. 212–219. doi:10.1145/237814.237866.

[3] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, D. Stehle, CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM, in: 2018 IEEE European Symposium on Security and Privacy (EuroS P), 2018, pp. 353–367. doi:10.1109/EuroSP.2018.00032.

[4] J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, F. Vercauteren, Saber: Module-LWR Based Key Exchange, CPA-Secure Encryption and CCA-Secure KEM, in: A. Joux, A. Nitaj, T. Rachidi (Eds.), Progress in Cryptology – AFRICACRYPT 2018, Springer International Publishing, Cham, 2018, pp. 282–305.

[5] S. Akleylek, K. Seyhan, Module learning with rounding based key agreement scheme with modified reconciliation, Computer Standards & Interfaces 79 (2022) 103549. doi:10.1016/j.csi.2021.103549.

[6] Manuel Clavel and Francisco Durán and Steven Eker and Patrick Lincoln and Narciso Martí-Oliet and José Meseguer and Carolyn L. Talcott (Ed.), All About Maude, volume 4350 of *Lecture Notes in Computer Science*, Springer, 2007. doi:10.1007/978-3-540-71999-1.

[7] D. Dolev, A. C. Yao, On the security of public key protocols, IEEE Trans. Inf. Theory 29 (1983) 198–207. doi:10.1109/TIT.1983.1056650.

[8] B. Blanchet, Security protocol verification: Symbolic and computational models, in: Principles of Security and Trust - First International Conference, POST 2012, volume 7215 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 3–29. doi:10.1007/978-3-642-28641-4_2.

[9] B. Blanchet, Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif, in: Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures, volume 8604 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 54–87. doi:10.1007/978-3-319-10082-1_3.

[10] S. Escobar, C. Meadows, J. Meseguer, A Rewriting-Based Inference System for the NRL Protocol Analyzer and Its Meta-Logical Properties, Theor. Comput. Sci. 367 (2006) 162–202. doi:10.1016/j.tcs.2006.08.035.

[11] B. Schmidt, S. Meier, C. Cremers, D. A. Basin, Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties, in: 25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012, IEEE Computer Society, 2012, pp. 78–94. doi:10.1109/CSF.2012.25.

[12] C. J. F. Cremers, The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols, in: A. Gupta, S. Malik (Eds.), Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings, volume 5123 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 414–418.

doi:10.1007/978-3-540-70545-1_38.

[13] R. Küsters, T. Truderung, Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation, in: Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009, IEEE Computer Society, 2009, pp. 157–171. doi:10.1109/CSF.2009.17.

[14] B. Blanchet, M. Abadi, C. Fournet, Automated verification of selected equivalences for security protocols, J. Log. Algebraic Methods Program. 75 (2008) 3–51. doi:10.1016/j.jlap.2007.06.002.

[15] S. Escobar, C. A. Meadows, J. Meseguer, State Space Reduction in the Maude-NRL Protocol Analyzer, in: Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings, volume 5283 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 548–562. doi:10.1007/978-3-540-88313-5_35.

[16] S. Escobar, J. Hendrix, C. A. Meadows, J. Meseguer, Diffie-Hellman Cryptographic Reasoning in the Maude-NRL Protocol Analyzer, in: Proceeding 2nd International Workshop on Security and Rewriting Techniques, 2006. URL: http://www.dsic.upv.es/~sescobar/papers/EscMeaMes-SecReT07.pdf.

[17] B. Schmidt, R. Sasse, C. Cremers, D. A. Basin, Automated verification of group key agreement protocols, in: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014, IEEE Computer Society, 2014, pp. 179–194. doi:10.1109/SP.2014.19.

[18] J. Donenfeld, K. Milner, Formal verification of the Wire- Guard protocol, 2018. URL: https://www.wireguard.com/papers/wireguard-formal-verification.pdf.

[19] V. K. Yadav, S. Venkatesan, S. Verma, Man in the Middle Attack on NTRU Key Exchange, in: S. Verma, R. S. Tomar, B. K. Chaurasia, V. Singh, J. Abawajy (Eds.), Communication, Networks and Computing, Springer Singapore, Singapore, 2019, pp. 251–261.

[20] V. Shoup, Sequences of games: a tool for taming complexity in security proofs, IACR Cryptol. ePrint Arch. (2004) 332. URL: http://eprint.iacr.org/2004/332.

[21] B. Blanchet, A computationally sound mechanized prover for security protocols, IEEE Trans. Dependable Secur. Comput. 5 (2008) 193–207. doi:10.1109/TDSC.2007.1005.

[22] J. Alwen, B. Blanchet, E. Hauck, E. Kiltz, B. Lipp, D. Riepel, Analysing the HPKE standard, in: Advances in Cryptology - EUROCRYPT 2021, volume 12696 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 87–116. doi:10.1007/978-3-030-77870-5_4.

[23] M. Campagna, E. Crockett, Hybrid Post-Quantum Key Encapsulation Methods (PQ KEM) for Transport Layer Security 1.2 (TLS), RFC, RFC Editor, 2021. URL: https://datatracker.ietf.org/doc/html/draft-campagna-tls-bike-sike-hybrid.