Formal specification and model checking of Saber lattice-based key encapsulation mechanism in Maude

Duong Dinh Tran*, Kazuhiro Ogata*, Santiago Escobar†, Sedat Akleylek‡ and Ayoub Otmani§
*Japan Advanced Institute of Science and Technology, Ishikawa 923-1292, Japan
Email: {duongtd, ogata}@jaist.ac.jp
†VRAIN, Universitat Politècnica de València, Valencia, Spain
Email: sescobar@upv.es
‡Ondokuz Mayis University, Samsun, Turkey
Email: sedat.akleylek@bil.omu.edu.tr
§University of Rouen Normandie, France
Email: ayoub.otmani@univ-rouen.fr

***Abstract*—The security of most public-key cryptosystems currently in use today is threatened by advances in quantum computing. 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. A large number 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 Saber lattice-based KEM. We first formally specify the mechanism in Maude, a rewriting logic-based specification/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*-KEM; Maude; post-quantum cryptography; lattice-based cryptography; model checking.**

I. Introduction

The most popular asymmetric (or public-key) primitives used today will become insecure under sufficient strong quantum computers running Shor's algorithm [1]. This is because the hard mathematical problems on which asymmetric primitives rely are hard only under conventional computers, but can be efficiently solved by a sufficient large-scale quantum 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, there are large numbers of proposals for post-quantum key encapsulation mechanisms (KEMs), which aim to securely establish a symmetric key between two parties. This is understandable because the key exchange algorithm can be said to be the most important building block of cryptosystems.

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 hard problem. The proof can be regarded as a mathematical reduction, such that it makes sure that the only chance to violate the security of such a construction is to solve the infeasible problem. However, computational 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.

We formally specify and model check Saber KEM [2]. In addition, Kyber [3] (precisely CRYSTALS-Kyber) and SK-MLWR [4] (the KEM proposed in [4] is called SK-MLWR in the present paper) are also tackled, but because of space limitation, they are not presented in this paper. We provide the specifications of them at https://github.com/duongtd23/kems-mc. Saber is a KEM whose security is based on the hardness of the Module Learning With Rounding (MLWR)

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.
DOI reference number: 10.18293/SEKE2022-097

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

problem, which belongs to lattice-based cryptography. We use Maude [5], a programming/specification language based on rewriting logic, to first formally specify the Dolev-Yao generic intruder [6] as well as these KEMs. By employing the Maude search command, a Man-In-The-Middle (MITM) attack is found. Although this kind of attack is not a novel attack for KEMs, the formal specifications 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 quantum secure primitives, such as Saber. 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 [7] 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 [8], Maude-NPA [9], and Tamarin [10]. 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 [11]. 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. [12] 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, [13] presented one case study with Diffie-Hellman key agreement protocol. Tamarin [10] is another tool for symbolic security verification of cryptographic protocols. 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.

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 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 hard problem (e.g., discrete logarithm, integer factorization). CryptoVerif [14] is a tool for mechanizing proof in the computational model. It can generate proofs by sequences of games automatically or with little user interaction. Alwen et al. [15] 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.

II. Saber Key Encapsulation Mechanism

A. Key encapsulation mechanism

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

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

A KEM is ϵ-*correct* if for all $(pk, sk) \leftarrow \mathrm{KeyGen}()$ and $(c, k) \leftarrow \mathrm{Encaps}(pk)$, it holds that $\Pr[\mathrm{Decaps}(c, sk) \neq k] \leq \epsilon$. We say it is *correct* if $\epsilon = 0$.

B. Saber

Let $\mathbb{Z}_q$ denote the ring of integers modulo q. Let 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. Single polynomials are written without markup, bold lower-case letters represent vectors with coefficients in R or R_q, and bold upper-case letters denote matrices. If $\mathcal{X}$ is a probability distribution over a set S, then $x \leftarrow \mathcal{X}$ denotes sampling $x \in S$ according to $\mathcal{X}$. $\mathcal{U}$ denotes the uniform distribution and β_μ is a centered binomial distribution with parameter μ (the samples are in $[-\mu/2, \mu/2]$). $\ll$ and $\gg$ are bitwise shift operations, and when they are used with polynomials and matrices, they are applied to each coefficient. $\mathcal{F}, \mathcal{G}, \mathcal{H}$ are hash functions that are used in Saber. gen is a function that generates a pseudorandom matrix $\mathbf{A} \in R_q^{l \times l}$ from a seed $seed_{\mathbf{A}}$.

Fig. 1 describes the three algorithms (KeyGen, Encaps, Decaps) of Saber.KEM. It employs the three algorithms (KeyGen, Enc, Dec) of Saber.PKE, which are shown in Fig. 2. Note that $\mathbf{h}$, h_1, and h_2 are constants; while ϵ_q, ϵ_p, ϵ_T, and μ receive different values on different security levels. The possible values for all of them can be found in [2]. Let us suppose that Alice performs KEM.KeyGen step and sends a public key pk to Bob. Upon receiving pk, Bob randomly chooses a m, performs KEM.Enc step, and sends back to Alice a ciphertext c. Upon receiving c, Alice performs KEM.Dec step, and computes the value of c'. With a very high probability c' is equal to c, implying that m' on

Alice's side equals m on Bob's side with an overwhelming probability. After that, they can derive the same key K.

KEM.KeyGen()
$(seed_{\mathbf{A}}, \mathbf{b}, \mathbf{s}) = \text{PKE.KeyGen}()$
$pk = (seed_{\mathbf{A}}, \mathbf{b})$
$pkh = \mathcal{F}(pk)$
$z = \mathcal{U}(\{0,1\}^{256})$
$sk = (z, pkh, pk, \mathbf{s})$
return (pk, sk)

$\xrightarrow{pk}$

KEM.Enc(pk)
$m \leftarrow \mathcal{U}(\{0,1\}^{256})$
$(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
$c = \text{PKE.Enc}(pk, m; r)$
$K = \mathcal{H}(\hat{K}, c)$
return (c, K)

$\xleftarrow{c}$

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

Figure 1. Saber.KEM

PKE.KeyGen()
$seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\mathbf{A} = \text{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
$r = \mathcal{U}(\{0,1\}^{256})$
$\mathbf{s} = \beta_\mu(R_q^{l \times 1}; r)$
$\mathbf{b} = ((\mathbf{A}^T\mathbf{s} + \mathbf{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
return $(pk := (seed_{\mathbf{A}}, \mathbf{b}), \mathbf{s})$

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

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

Figure 2. Saber.PKE

III. Formal specification of Saber

A. Formalization of polynomials, vectors, and matrices

We first introduce sort `Poly` that represents polynomials:

```
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 _md_ : Poly Nat -> Poly [ctor prec 32] .
op _p-_ : Poly Poly -> Poly [prec 33] .
op neg_ : Poly -> Poly [ctor] .
```

where `Int` and `Nat` are sorts of integers and natural numbers, respectively. 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, while `md` denotes the modulo operation. `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`). Let `P1`, `P2`, and `P3` be Maude variables of `Poly`. We define 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+ neg(P1) = 0 .        eq neg(neg(P1)) = P1 .
eq P1 p- P2 = P1 p+ neg(P2) .
eq neg(P1 p+ P2) = neg(P1) p+ neg(P2) .
eq neg(P1 md K) = neg(P1) md K .
```

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 Maude 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) .
```

B. Formalization of honest parties

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

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

where `Prin` and `Msg` are sorts denoting principals and messages, respectively. `PVPair` is the sort of polynomial and vector pairs. `MState` 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 of `msg1` and `msg2` 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.

We model the network as an AC-collection of messages that can be used by the intruder as his/her storage. Consequently, the empty network (i.e., the empty collection) 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. In this paper, a state is expressed as an AC-collection of name-value pairs called observable components. To formally specify Saber in Maude, we use the following observable components:

- (`nw` : $msgs$) - $msgs$ is the AC-collection of messages in the network;

- (keys[p] : $keys$) - $keys$ is an AC-collection 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 mechanism;
- (seed[p] : sd) - sd is the random seed $seed_{\mathbf{A}}$ (used in Fig. 2) of principal p;
- (r[p] : r_0) - r_0 is the random seed r (used in Fig. 2) of principal p;
- (m[p] : m_0) - m_0 is the random seed m (shown in Fig. 1) of principal p;
- (rd-seed : rds) - rds is a list of available values as the random seed $seed_{\mathbf{A}}$ (we use list, but not set, to reduce the state space for searching). Each time when a principal queries for a random value of $seed_{\mathbf{A}}$, the top value in rds is removed and returned to the principal;
- (rd-r : $rdrs$) - $rdrs$ is a list of the available values as random seed r;
- (rd-m : $rdms$) - $rdms$ is a list of the available values as random seed m;
- (glean-keys : $gkeys$) - $gkeys$ is the AC-collection of shared keys gleaned by the intruder;
- (seeds : sds) - sds is the collection of the random seeds $seed_{\mathrm{A}}$ used by the intruder;
- (rs : rs) - rs is the collection of the random seeds r used by the intruder;
- (ms : ms) - ms is the collection of random seeds m used by the intruder;

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

```
{(nw: empty) (keys[alice]: empty) (keys[bob]: empty)
(prins: (alice ; bob ; eve)) (rd-seed: (seed1, seed2))
(rd-r: (r1, r2)) (rd-m: (m1, m2)) (glean-keys: empty)
(seed[alice]: 0) (seed[bob]: 0) (m[alice]: 0)
(m[bob]: 0) (seeds: empty) (rs: empty) (ms: empty)}
```

With the honest parties, we specify three transitions: keygen, encaps, and decaps, which correspond to the three steps of the mechanism. We declare Maude constants esp, esq, esT, p, q, h1, and h to denote ϵ_p, ϵ_q, ϵ_T, p, q, h_1, and $\mathbf{h}$, respectively. Let OCs be a Maude variable of observable component collections, A, B, and C be Maude variables of principals (possibly intruder), and PS be a Maude variable of principal collections. Let SD, R, P1, P2, and M, be Maude variables of polynomials; PL and PL2 be Maude variables of polynomial lists. Let G, F, and H denote the hash functions $\mathcal{G}$, $\mathcal{F}$, and $\mathcal{H}$, respectively. Let MS be a Maude variable of networks. The rewrite rule keygen is defined as follows:

```
crl [keygen] : {(rd-seed: (SD, PL)) (rd-r: (R, PL2))
(seed[A]: P1) (r[A]: P2) (prins: (A ; B ; PS))
(nw: MS) OCs}
=> {(rd-seed: PL) (rd-r: PL2)
(seed[A]: SD) (r[A]: R) (prins: (A ; B ; PS))
(nw: (MS ; msg1(A,A,B, pvPair(SD, VB), sent))) OCs}
if MA := gen-A(SD)   /\   S := gen-s(R) /\
VB := shiftRV((tp(MA) m* S v+ h) mdv q, esq - esp) .
```

where MA and VB are Maude variables of polynomial matrices and vectors. tp, shiftRV, and mdv denote matrix transpose, vector bitwise right shift, and vector modulo operations. gen-A and gen-s denote the function gen and the sampling procedures, outputting the matrix $\mathbf{A}$ and the vector $\mathbf{s}$, respectively. The rewrite rule says that when there exist a polynomial SD in rd-seed and a polynomial R in rd-r, A picks it as a random seed r, builds a message msg1 exactly following the KeyGen step, and sends it to B. seed[A] and r[A] are set to SD and R, respectively, and the two values are removed from rd-seed and rd-r.

The rewrite rule encap is defined as follows:

```
crl [encaps] : {(rd-m: (M, PL)) (m[B]: P1) (keys[B]: KS)
(nw: (msg1(C,A,B, pvPair(SD,VB), sent) ; MS)) OCs}
=> {(keys[B]: (KS ; key( H(1st(Kr), CB) , A)))
(nw: (msg1(C,A,B, pvPair(SD, VB), replied) ;
msg2(B,B,A, CB, sent) ; MS)) (rd-m: PL) (m[B]: M)  OCs}
if Kr := G(F(pvPair(SD,VB)), M) /\
   CB := enc(SD,VB,M,2nd(Kr)) .
```

where KS is a Maude variable representing a collection of shared keys, Kr is a Maude variable denoting a pair of polynomials, in which 1st and 2nd are its projection operators. Following the PKE.Enc$(pk, m; r)$ in Fig. 2, enc is defined as follows:

```
ceq enc(SD, VB, M, R) = pvPair(CM, VB')
if MA := gen-A(SD) /\  S' := gen-s(R) /\
VB' := shiftRV((MA m* S' v+ h) mdv q, esq - esp) /\
V' := tpV(VB) dot S' /\
CM := shiftR(((V' p+ h1) p- (2 ^ (esp - 1)) p* M) md p,
       esp - esT) .
```

where tpV(VB) denotes the transpose vector of VB and shiftR denotes the polynomial bitwise right shift. 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, and 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 can be defined likewise. Note that we only consider the overwhelming case, i.e., Alice successfully recovers m in Decaps step. We assume that the error tolerance gaps made by error components always be silent, making m' equal m. To this end, we need to define some properties of the bitwise shift operation and polynomials. The first property is as follows: $(2^{\epsilon_p - \epsilon_T} c_m \gg (\epsilon_p - \epsilon_T)) \approx c_m$. It is specified by the following equation:

```
ceq 2PT p* shiftR(CM, PT) = CM
if PT := esp - esT  /\  2PT := 2 ^ PT .
```

where PT and 2PT are variables of integers.

If all coefficients of $\mathbf{s}$ and $\mathbf{s}'$ are small in comparison with p and q, then we have the second property as follows: $(((\mathbf{A}\mathbf{s}' +$

$\mathbf{h}) \bmod q) \gg (\epsilon_q - \epsilon_p))^T \mathbf{s} \approx (((\mathbf{A}^T\mathbf{s} + \mathbf{h}) \bmod q) \gg (\epsilon_q - \epsilon_p))\mathbf{s}'$. The property is specified by the following equation:

```
ceq tpV(shiftRV( (MA m* S' v+ h) mdv q, esq - esp ))
dot S p+ neg(tpV(shiftRV( (tp(MA) m* S v+ h) mdv q,
esq - esp )) dot S') = 0
if isSmall?(S) and isSmall?(S') .
```

where `isSmall?` is a predicate, returning true if all coefficients of `S` (or `S'`) are small in comparison with q. Note that the result of `gen-s` is always defined to be "small," which is done by the following equation: `eq isSmall?(gen-s(R)) = true .`

Finally, to specify the property $((h_2 - h_1 + 2^{\epsilon_p - 1}m) \bmod p) \gg (\epsilon_p - 1) \approx m$, we introduce the following equation:

```
ceq shiftR( (h2 p+ neg(h1) p+ 2P1 p* M) md p, esp - 1)
= M if 2P1 := 2 ^ (esp - 1) .
```

C. 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] : {(seeds: (SD ; PC1)) (rs: (R ; PC2))
(nw: (msg1(A,A,B, pvPair(SD-A,VB-A), sent) ; MS)) OCs}
=> {(seeds: (SD ; PC1)) (rs: (R ; PC2))
(nw: (msg1(A,A,B, pvPair(SD-A,VB-A), intercepted) ;
msg1(eve,A,B, pvPair(SD, VB), sent) ; MS)) OCs}
if MA := gen-A(SD)   /\   S := gen-s(R) /\
VB := shiftRV((tp(MA) m* S v+ h) mdv q, esq - esp) .
```

where `PC1` and `PC2` are Maude variables of polynomial collections. 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 available values `SD` and `R` for the random seeds $seed_{\mathbf{A}}$ and r. These two kinds of random values cannot be gleaned from the network, but `eve` can only construct them by randomly choosing a new value as the rewrite rule `build-ds` as follows:

```
rl [build-sds] : {(rd-seed: (SD, PL)) (seeds: PC1) OCs}
=> {(rd-seed: PL) (seeds: (SD ; PC1)) OCs} .
rl [build-rs] : {(rd-r: (R, PL)) (rs: PC2) OCs}
=> {(rd-r: PL) (rs: (R ; PC2)) OCs} .
```

There are two more rewrite rules `encaps-eve` and `decaps-eve` to specify the intruder's behavior. `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`.

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

We introduce the following search command:

Alice.Step-1
$(seed_{\mathbf{A}}, \mathbf{b}, \mathbf{s}) = \text{PKE.KeyGen}()$
$pk = (seed_{\mathbf{A}}, \mathbf{b})$
$pkh = \mathcal{F}(pk)$
return $(pk, sk := (phk, pk, \mathbf{s}))$

Eve.Step-2$(pk = (seed_{\mathbf{A}}, \mathbf{b}))$
$(seed_{\mathbf{A}e}, \mathbf{b}_e, \mathbf{s}_e) = \text{PKE.KeyGen}()$
$pk_e = (seed_{\mathbf{A}e}, \mathbf{b}_e)$
$pkh_e = \mathcal{F}(pk_e)$
return $(pk_e, sk_e := (phk_e, pk_e, \mathbf{s}_e))$

Bob.Step-3$(pk_e = (seed_{\mathbf{A}e}, \mathbf{b}_e))$
$m \leftarrow \mathcal{U}(\{0,1\}^{256})$
$(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk_e), m)$
$c = \text{PKE.Enc}(pk_e, m; r)$
return $(c, K_b := \mathcal{H}(\hat{K}, c))$

Eve.Step-4$(pk = (seed_{\mathbf{A}}, \mathbf{b}))$
$m_e \leftarrow \mathcal{U}(\{0,1\}^{256})$
$(\hat{K}_e, r_e) = \mathcal{G}(\mathcal{F}(pk), m_e)$
$c_e = \text{PKE.Enc}(pk, m_e; r_e)$
return $(c_e, K_a := \mathcal{H}(\hat{K}_e, c_e))$

Alice.Step-5$(c_e, sk := (phk, pk, \mathbf{s}))$
$m' = \text{PKE.Dec}(\mathbf{s}, c_e)$
$(\hat{K}', r') = \mathcal{G}(pkh, m')$
$c' = \text{PKE.Enc}(pk, m'; r')$
if $c' = c_e$ **then return** $K_a := \mathcal{H}(\hat{K}', c')$

Eve.Step-6$(c, sk_e := (phk_e, pk_e, \mathbf{s}_e))$
$m'_e = \text{PKE.Dec}(\mathbf{s}_e, c)$
$(\hat{K}'_e, r'_e) = \mathcal{G}(pkh_e, m'_e)$
$c'_e = \text{PKE.Enc}(pk_e, m'_e; r'_e)$
if $c = c'_e$ **then return** $K_b := \mathcal{H}(\hat{K}'_e, c)$

Figure 3. A counterexample found by Maude

```
search [1] in Saber : 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 Maude variables that denote arbitrary shared keys. `K1` may or may not equal `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 Saber, which is visualized from the path leading to the counterexample Maude returned. There are mainly six 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 send pk to Bob.

Step-2: Eve intercepts the first message sent from Alice to Bob. She follows the KEM.KeyGen step to generate a pair (pk_e, sk_e), impersonating Alice to send pk_e to Bob.

Step-3: Bob receives pk_e thinking it is from Alice. As a response, he takes a random m, performs KEM.Enc with the input 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 the ciphertext c sent from Bob to Alice. Then, she takes a random m_e, performs PKE.Enc with inputs pk and m_e, 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 with inputs c_e and sk, obtains the shared key K_a. She believes that K_a is the shared key obtained by her and Bob.

Step-6: Eve performs KEM.Dec with inputs c and sk_e, and obtains the shared key K_b.

The reachable state space in the experiment is finite. Indeed, if we try to run the following command: `search in SABER : init =>* {OCs} .`, the number of returned solutions is finite, implying that the state space is finite. This can be understandable. The key point is that the numbers of possible values that each observable component (i.e., a name-value pair) can receive are finite.

Remark. Readers may argue that this kind of attack is not a novel attack because Saber KEM does not go along with any solution for authentication. We agree on it. The paper instead illustrates one symbolic approach for reasoning 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 quantum-resistant 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.

V. Conclusion

The paper has presented an approach to security analysis of Saber.KEM in the symbolic model. We first used Maude as a specification language to formally specify the mechanism. After that, by employing Maude search command, an MITM attack was found. The occurrence of the attack is basically because a KEM alone does not come with an authentication solution.

Amazon Web Service team has proposed a post-quantum TLS protocol [16] 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. To this end, the most important task is to come up with a new intruder model because intruders will be able to utilize quantum computers on which quantum algorithms, such as Shor's one [1], run in the post-quantum era.

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.

[2] 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 *AFRICACRYPT 2018*, 2018, pp. 282–305.

[3] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS - kyber: A CCA-Secure Module-Lattice-Based KEM," in *2018 IEEE EuroS&P*, 2018, pp. 353–367.

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

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, Eds., *All About Maude*, 2007, vol. 4350.

[6] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. 29, no. 2, pp. 198–207, 1983.

[7] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *POST 2012, ETAPS 2012*, vol. 7215. Springer, 2012, pp. 3–29.

[8] ——, "Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif," in *FOSAD 2012/2013 Tutorial Lectures*, vol. 8604, 2013, pp. 54–87.

[9] S. Escobar, C. Meadows, and J. Meseguer, "A Rewriting-Based Inference System for the NRL Protocol Analyzer and Its Meta-Logical Properties," *Theor. Comput. Sci.*, vol. 367, no. 1, p. 162–202, Nov. 2006.

[10] B. Schmidt, S. Meier, C. Cremers, and D. A. Basin, "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties," in *IEEE CSF 2012*, 2012, pp. 78–94.

[11] R. Küsters and T. Truderung, "Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation," in *IEEE CSF 2009*, 2009, pp. 157–171.

[12] S. Escobar, C. A. Meadows, and J. Meseguer, "State Space Reduction in the Maude-NRL Protocol Analyzer," in *ESORICS 2008*, vol. 5283, 2008, pp. 548–562.

[13] S. Escobar, J. Hendrix, C. A. Meadows, and J. Meseguer, "Diffie-Hellman Cryptographic Reasoning in the Maude-NRL Protocol Analyzer," in *Proceeding 2nd International Workshop on Security and Rewriting Techniques*, 2006.

[14] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Trans. Dependable Secur. Comput.*, vol. 5, no. 4, pp. 193–207, 2008.

[15] J. Alwen, B. Blanchet, E. Hauck, E. Kiltz, B. Lipp, and D. Riepel, "Analysing the HPKE standard," in *EUROCRYPT 2021*, vol. 12696, 2021, pp. 87–116.

[16] M. Campagna and E. Crockett, "Hybrid Post-Quantum Key Encapsulation Methods (PQ KEM) for Transport Layer Security 1.2 (TLS)," RFC Editor, RFC, 09 2021.