Hybrid Post-Quantum TLS formal specification in Maude-NPA - toward its security analysis*

Duong Dinh Tran[1,*,†], Canh Minh Do[1,†], Santiago Escobar[2] and Kazuhiro Ogata[1]

[1]Japan Advanced Institute of Science and Technology, Ishikawa 923-1292, Japan
[2]VRAIN, Universitat Politècnica de València, Valencia, Spain

Abstract

This paper presents a formal specification of the Hybrid Post-Quantum TLS protocol in Maude-NPA, toward a security analysis of the protocol, where Hybrid Post-Quantum TLS is a quantum-resistant version of TLS proposed by AWS as a preparation against future attacks from quantum computers. The proposed protocol uses a hybrid key exchange mode: one is a classical key exchange algorithm and the other is a post-quantum key encapsulation mechanism. One of our assumptions about the intruder's capabilities is that the intruder can break the security of the classical key exchange algorithm by utilizing the power of large-scale quantum computers. In the present paper, we focus on presenting how to formally specify the protocol in Maude-NPA, which is a well-known tool for analyzing the security of cryptographic protocols, so that later on we can conduct the formal analysis of the protocol with some security properties.

Keywords

post-quantum, TLS, Maude-NPA, formal specification, security analysis

1. Introduction

Post-quantum cryptographic protocols refer to those alternatives to classical cryptographic protocols in order to oppose potential attacks from quantum computers. Research on quantum computers, which exploits quantum mechanical phenomena to solve hard mathematical problems that are intractable for traditional computers, has increased significantly in recent years. For example, the integer factorization problem is no longer hard under large-scale quantum computers running Shor's algorithm [1]. That leads to most of the asymmetric primitives used today will become insecure under sufficiently powerful quantum computers because the computationally hard mathematical problems on which they are relying (i.e., the integer factorization

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

*D. D. Tran, C. M. Do, and K. Ogata have been supported by JST SICORP Grant Number JPMJSC20C2, Japan. 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.
†These authors contributed equally.
duongtd@jaist.ac.jp (D. D. Tran); canhdominh@jaist.ac.jp (C. M. Do); sescobar@upv.es (S. Escobar); ogata@jaist.ac.jp (K. Ogata)

problem, the discrete logarithm problem, and the elliptic-curve discrete logarithm problem) can be efficiently solved by a sufficiently large quantum computer. Meanwhile, many important milestones in the construction of quantum computer hardware have been achieved with the involvement of many giants, such as Intel, IBM, and Google. That makes practical quantum computers closer and closer to reality. Therefore, research on building new post-quantum cryptographic protocols and security verification of those post-quantum cryptosystems has got extensive attention from cryptography and security research groups in recent years.

The Hybrid Post-Quantum Transport Layer Security (TLS) Protocol [2] has been proposed by Amazon Web Services (AWS) as a quantum-resistant version of the TLS 1.2 protocol, where TLS is known as one of the most widely used cryptographic protocols. The *hybrid* terminology in the name of the proposed protocol refers to the hybrid key exchange mode used in the protocol: one is a conventional key exchange algorithm, i.e., Elliptic Curve Diffie-Hellman (ECDH), and the other is a post-quantum key encapsulation mechanism (KEM), which can be one of Kyber [3], SIKE [4], and BIKE [5]. That results in a shared secret key at least as secure as ECDH against a classical adversary and at least as secure as the selected post-quantum KEM (PQ KEM) against a quantum adversary.

Maude-NPA is a powerful formal verification tool for analyzing cryptographic protocols that uses a backward narrowing reachability analysis modulo an equational theory and the Dolev-Yao strand space model [6, 7], which gives intruders capable of intercepting, modifying, and injecting messages to impersonate other protocol principals. Narrowing is a generalization of term rewriting that allows logical variables in terms and replaces pattern matching by unification and so Maude-NPA supports a symbolic execution. The backward narrowing reachability analysis starts from a final insecure pattern that represents insecure states, a so-called attack pattern, to check whether it is reachable from an initial state, which has no further backward steps. If that is the case, the attack concerned can be conducted for the protocol under verification; otherwise, the attack cannot. The advantage of Maude-NPA is that it supports protocols with an unbounded session model and different equational theories as other modern analyzers, such as Tamarin [8], and especially it is fully automatic. This paper focuses on presenting the formal specification of the Hybrid Post-Quantum TLS protocol in Maude-NPA, which is the first step toward conducting a security formal analysis of the protocol.

The remaining of this paper is organized as follows. Section 2 first gives some preliminaries related to Maude and Maude-NPA. Then, Section 3 describes in detail messages exchanged in the Hybrid PQ TLS protocol. Section 4 is the main content of the paper, where we present how to formally specify the protocol in Maude-NPA. After that, Section 5 mentions some related work, and finally, Section 6 summarizes our paper. The full specification of the protocol in Maude-NPA is publicly available at https://github.com/duongtd23/PQTLS-MaudeNPA.

2. Preliminaries

Maude-NPA is implemented in Maude [9], a declarative language and high-performance tool that focuses on simplicity, expressiveness, and performance to support the formal specification and analysis of concurrent programs/systems in rewriting logic. Maude can directly specify order-sorted equational logics and rewriting logic [10], and the tool provides several formal

analysis methods, such as reachability analysis and LTL model checking. This section gives the syntax of the Maude language in a nutshell (see [9] for more detail) and describes how narrowing works with an example.

Functional modules

A functional module $\mathcal{M}$ specifies an order-sorted equational logic theory (Σ, E) with the syntax: **fmod** $\mathcal{M}$ **is** (Σ, E) **endfm**, where Σ is an order-sorted signature and E is the collection of equations in the functional module. (Σ, E) may contain a set of declarations as follows:

- importations of previously defined modules (**protecting** ... or **extending** ... or **including** ...)
- declarations of sorts (**sort** s . or **sorts** s s' .)
- subsort declarations (**subsort** s < s' .)
- declarations of function symbols (**op** f : s_1 ... $s_n \rightarrow s$ [att_1 ... att_k] .)
- declarations of variables (**vars** v v' .)
- unconditional equations (**eq** t = t' .)
- conditional equations (**ceq** t = t' **if** $cond$.)

where $s, s_1, \ldots, s_n$ are sort names, v, v' are variable names, t, t' are terms, $cond$ is a conjunction of equations (e.g., t = t'), and $att_1, \ldots att_k$ are equational attributes. Equations are used as *equational rules* to perform the simplification in which instances of the lefthand side pattern that match subterms of a subject term are replaced by the corresponding instances of the righthand side. The process is called *term rewriting* and the result of simplifying a term is called its *normal form.*

System modules

A system module $\mathcal{R}$ specifies a rewrite theory (Σ, E, R) with the syntax: **mod** $\mathcal{R}$ **is** (Σ, E, R) **endm**, where Σ and E are the same as those in an equational theory and R is the collection of rewrite rules in the system module. (Σ, E, R) may contain all possible declarations in (Σ, E) and rewrite rules in R as follows:

- unconditional rewrite rules (**rl** [$label$] : u => v .)
- conditional rewrite rules (**crl** [$label$] : u => v **if** $cond$.)

where $label$ is the name of a rewrite rule, u, v are terms, and $cond$ is a conjunction of equations and/or rewrites (e.g., t => t'). Rewrite rules are also computed by rewriting from left to right modulo the equations in the system module and regarded as *local transition rules*, making many possible state transitions from a given state in a concurrent system.

Narrowing

Narrowing is a generalization of term rewriting that allows logical variables in terms and replaces pattern matching by unification. Let us use a classical example in the Maude community to describe how narrowing works. The formal definition of narrowing can be found in [11]. The following system module specifies a concurrent machine to buy cakes (`c`) and apples (`a`) with dollars (`$`) and quarters (`q`). We suppose that a cake costs a dollar (see the rewrite rule labeled as `buy-c` below) while an apple costs three quarters (the rewrite rule `buy-a`). The machine only allows buying cakes and apples with dollars. However, the machine can change four quarters into a dollar (the rewrite rule `change`).

```
mod NARROWING-VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op __ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op __ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops c a : -> Item .
  var M : Marking .
  rl [buy-a] : < M $ > => < M a q > [narrowing] .
  rl [buy-c] : < M $ > => < M c > [narrowing] .
  eq [change] : q q q q M = $ M [variant] .
endm
```

where the `__` operator is associative and commutative and has an identity element `empty`, the `<_>` operator specifies the machine state, and `narrowing` and `variant` attributes are specially used for narrowing and variant-based equational unification algorithms [12]. Let us consider a term `< M1 >` as an initial state that only contains a variable of the sort `Money`. There would be several traces from the initial state by using narrowing. At each narrowing step, we must choose which subterm of the subject term, which rewrite rule of the specification, and which instantiation on the variables of the subterm and the left-hand side of the rewrite rule (or which unifier (or substitution) of the subterm and the left-hand side of the rewrite rule) are going to be considered. Note that only rewrite rules with a `narrowing` attribute are considered and only equations with a `variant` attribute are used to decide the unification problem modulo the equations. Each narrowing step applied to a given state produces a new branch in the reachability tree. For example, for each rewrite rule of the machine, there is only one unifier that makes the initial state equal to the left-hand side of the rewrite rule. Therefore, we can only obtain the following two narrowing steps, generating only two successor states from the initial state, by performing narrowing just by one step as follows:

$$\texttt{< M1 >} \rightsquigarrow_{\sigma_1,\ \texttt{buy-a}} \texttt{< a q M2 >}$$
$$\texttt{< M1 >} \rightsquigarrow_{\sigma_1',\ \texttt{buy-c}} \texttt{< c M2' >}$$

where `M2` and `M2'` are variables of the sort `Money` and the substitutions are $\sigma_1 = \{\texttt{M1} \mapsto \texttt{\$ M2},\ \texttt{M} \mapsto \texttt{M2}\}$ and $\sigma_1' = \{\texttt{M1} \mapsto \texttt{\$ M2'},\ \texttt{M} \mapsto \texttt{M2'}\}$ with the rewrite rules `buy-a` and `buy-c`, respectively. Note that `M` in the substitutions is the variable used in the left-hand side of the rewrite rules. If

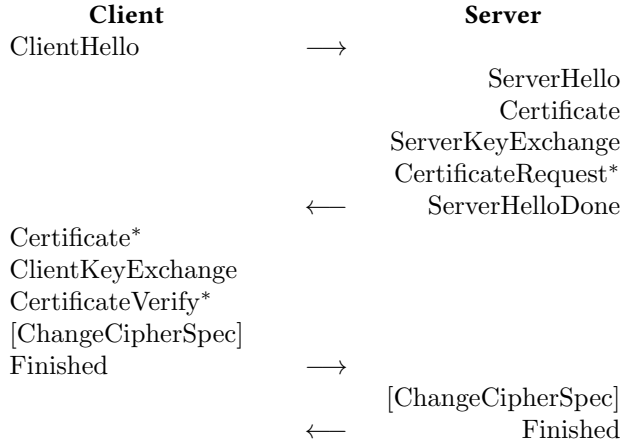

Figure 1: Messages exchanged in a full handshake of the Hybrid Post-Quantum TLS protocol [2]

we take the successor state `< a q M2 >` and perform two more consecutive narrowing steps, it makes one trace taking us to the state `< a a c q M4 >`, which also contains a variable `M4` of the sort `Money`. The narrowing sequence associated to the state is as follows:

$$\texttt{< M1 >} \rightsquigarrow_{\sigma_1,\ \texttt{buy-a}} \texttt{< a q M2 >} \rightsquigarrow_{\sigma_2,\ \texttt{buy-c}} \texttt{< a c q M3 >} \rightsquigarrow_{\sigma_3,\ \texttt{buy-a}} \texttt{< a a c q M4 >}$$

where `M3` and `M4` are variables of the sort `Money` and the substitutions are $\sigma_2 = \{\texttt{M2} \mapsto \texttt{\$ M3},\ \texttt{M} \mapsto \texttt{a q M3}\}$ and $\sigma_3 = \{\texttt{M3} \mapsto \texttt{q q q M4},\ \texttt{M} \mapsto \texttt{a c M4}\}$ with the rewrite rules `buy-c` and `buy-a`, respectively. In the third narrowing step, when we apply the substitution σ_3, the instance of `< a c q M3 >` is `< a c q q q q M4 >`, while the instance of the left-hand side of the rewrite rule `buy-a` is `< a c M4 $ >`. The two instances are actually equal thanks to the commutative property and the equation `change`. Therefore, the rewrite rule `buy-a` modulo the equational theory is used to obtain the state `< a a c q M4 >`. By using narrowing, we can solve the reachability problem $St \rightsquigarrow_{R,E} St'$ where St and St' are patterns (terms that may have variables) of the sort `State` such that some conditions are satisfied, and R, E are the rewrite rules and equations in the specification.

3. Hybrid Post-Quantum TLS 1.2

Figure 1 graphically shows messages exchanged in a full handshake of the Hybrid Post-Quantum TLS protocol. In this figure, * indicates that the message is not sent unless client authentication is requested, while [ ] indicates that the message actually belongs to the change cipher spec protocol. The hybrid key exchange mechanism in the proposed protocol directly impacts on ClientHello, ServerHello, ServerKeyExchange, and ClientKeyExchange messages.

The messages exchanged in the protocol can be described as follows. Initially, a client sends a ClientHello message to a server to start a new session. In the full handshake mode, a ClientHello message consists of the protocol version, a random number, an empty session ID, a list of cipher suites (which contains combinations of cryptographic options supported by the client), and a set of post-quantum KEM parameters (including the name of KEM and its

parameters) supported by the client. Upon reception of the ClientHello message, the server sends a ServerHello message back to the client, which consists of the protocol version, a random number, a non-empty session ID, and a selected cipher suite. Together with that message, the server also sends their digital certificate (a Certificate message) and a ServerKeyExchange message to the client. The ServerKeyExchange message contains the server's ECDHE & PQ KEM public keys and a signature over the two public keys together with the two random numbers in the ClientHello and ServerHello messages signed by the server's long-term private key. In the case when client authentication is requested, the server sends a CertificateRequest message, which is optional. A ServerHelloDone message is then sent to the client to signal the completion of the hello-message phase on the server side.

Upon reception of the ServerHelloDone message, the client replies to the server with a ClientKeyExchange message, which consists of the client's ECDHE public key and the KEM ciphertext. Before that, if the server has sent a CertificateRequest message, the client must send their certificate (a Certificate message) first. Similarly, if the server has requested client authentication, the client will then send a CertificateVerify message, whose content is a digital signature over all handshake messages exchanged so far signed by the client's long-term private key. After that, a ChangeCipherSpec message (which belongs to the change cipher spec protocol) is sent to notify that all subsequent messages will be encrypted by the newly negotiated keys. Finally, the client sends a Finished message, whose content is a hash of all handshake messages encrypted by a handshake key just negotiated.

Upon reception of the Finished message from the client, the server must validate that the message is correct. If so, the server sends their own ChangeCipherSpec and Finished messages. Once the client receives the Finished message, it also has to validate that the message is correct. After that, both sides are ready for secure data communication.

4. Hybrid PQ TLS formal specification in Maude-NPA

Before going into detail about the formal specification of the Hybrid PQ TLS protocol, this section first presents *strands* notation, and how we can use them to model protocol execution via a simple example. Comprehension of this concept is an essential step to understanding how we can specify the protocol in Maude-NPA later.

4.1. Formal specification by strands

Maude-NPA uses *strands* [7] to specify the behavior of a protocol execution and the intruder capabilities. Each strand is a sequence of positive and negative messages describing a principal executing a protocol, or the intruder performing actions. A strand is in the following form:

$$:: r_1, \ldots, r_k :: [+(m_1), -(m_2), \ \ldots, -(m_i) \mid + (m_{i+1}), \ \ldots]$$

where $r_1, \ldots, r_k$ denote unique freshes generated in the strand, and a positive message $+(m)$ and a negative message $-(m)$ denote sending and receiving the message m, respectively. The vertical bar is used to distinguish between present and future when the strand appears in a state description. Messages appearing before the bar were sent/received in the past, while messages

appearing after the bar will be sent/received in the future. To illustrate how to specify protocol execution with strands, let us consider the Needham-Schroeder Public Key (NSPK) protocol [13], which has three following messages exchanged written in the standard Alice-and-Bob notation:

i) $A \rightarrow B : pk(B, A ; N_A)$
ii) $B \rightarrow A : pk(A, N_A ; N_B)$
iii) $A \rightarrow B : pk(B, N_B)$

where A and B denote Alice and Bob principal identifiers, N_A and N_B are nonces (unguessable values) generated by A and B, respectively, and $pk(A, m)$ denotes the encryption of message m by the public key of A. The three messages can be explained as follows. A first generates a nonce N_A and sends it together with their ID encrypted by B's public key to B (note that the semicolon denotes the concatenation). Upon receiving that message, B decrypts it and obtains a nonce. The nonce and a newly generated nonce N_B are encrypted by A 's public key and then sent back to A. When receiving the message, A decrypts it, getting two nonces, and checking if the first one is exactly the one that A has sent in this session. A finishes the communication by sending to B the other nonce encrypted under B's public key.

To model the protocol in Maude-NPA, we use some built-in sorts in Maude-NPA, such as the sort `Msg` that represents messages and the sort `Fresh` that is used to identify terms that must be unique. Besides, we introduce some sorts, such as `Name` and `Nonce` to distinguish principal names and nonces, respectively. The two sorts are sub sorts of the sort `Msg`, which are declared as follows:

```
sorts Name Nonce .
subsort Name Nonce < Msg .
```

`Nonce` is defined by the following operator:

```
op n : Name Fresh -> Nonce [frozen] .
```

where the `frozen` attribute is attached following the Maude-NPA convention, which is necessary to tell Maude not to attempt to apply rewrites at the arguments of this operator. Suppose that `A` and `r` respectively are variables of the sorts `Name` and `Fresh`, then `n(A,r)` denotes the nonce generated by `A`, where `r` guarantees its uniqueness. We also declare two public & private encryption operators respectively as follows:

```
op pk : Name Msg -> Msg [frozen] .
op sk : Name Msg -> Msg [frozen] .
```

Let `B` and `N` be variables of the sorts `Name` and `Nonce`, respectively. The strand specifying the protocol execution from the A side is then defined as follows:

```
:: r ::
[ nil | +(pk(B, A ; n(A,r))), -(pk(A, n(A,r) ; N)), +(pk(B, N)), nil ]
```

The strand says that initially, `A` generates a nonce based on the fresh `r` and sends it to `B` together with `A`'s ID encrypted by `B`'s public key. When `A` receives another message whose content is their nonce sent in the first message and another nonce `N` encrypted under `A`'s public key (which can be checked by using their secret key to decrypt the received ciphertext), `A` replies to `B` the new nonce `N` encrypted under `B`'s public key. Note that `:: r ::` denotes the fresh `r` is generated in

the strand and all messages are put after the vertical bar following the Maude-NPA convention because the vertical bar is irrelevant when specifying the protocol execution. Note also that `;` is the infix operator of messages concatenation, which is declared as follows:

```
op _;_ : Msg Msg -> Msg [frozen] .
```

In the same manner, the strand specifying the protocol execution from the B side can be defined as follows:

```
:: r ::
[ nil | -(pk(B, A ; N)), +(pk(A, N ; n(B,r))), -(pk(B, n(B,r))), nil ]
```

Strands are also used for the specification of the intruder capabilities. But in this case, such an intruder strand is limited to be in form of a sequence of negative messages (possibly empty) followed by one positive message combining all previous variables under a function symbol. For example, to specify the intruder capability in concatenating two arbitrary messages, we define the following strand:

```
:: nil :: [ nil | -(M1), -(M2), +(M1 ; M2), nil ]
```

where `M1` and `M2` are variables of the sort `Msg`. The strand says that if the messages `M1` and `M2` are available to the intruder, then the intruder can produce the message `M1 ; M2`.

4.2. Hybrid PQ TLS formal specification

4.2.1. KEM specification

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

- `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 an encapsulation (or ciphertext) c and a shared key $k \in \mathcal{K}$.
- `Decaps`(c, sk) $\rightarrow k$: A (usually deterministic) *decapsulation* algorithm that takes as inputs a ciphertext c and a secret key sk, and outputs a shared key $k \in \mathcal{K}$.

A KEM is ϵ-$correct$ if for all $(pk, sk) \leftarrow$ `KeyGen`() and $(c, k) \leftarrow$ `Encaps`(pk), it holds that:

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

In this paper, we assume that all KEMs are 0-$correct$, which means that `Encaps` and `Decaps` always correctly return the same shared key k. Idealizing assumptions like that are typically necessary when conducting security analysis in the symbolic model. Furthermore, because a `Decaps`-failure probability is really small, typically almost 0, we are not over-idealizing when omitting such `Decaps`-failure cases. For example, with Kyber, that failure probability is below 2^{-140} [3], i.e., Kyber is ϵ-$correct$ with $\epsilon < 2^{-140}$.

To model KEMs in Maude-NPA, we first introduce `PqSk`, `PqPk`, `Cipher`, and `PqKey` sorts to represent secret keys, public keys, encapsulations, and shared keys, respectively. The sort `PqSk` is specified as follows:

```
op pqSk : Name Fresh -> PqSk [frozen] .
```

The argument of the sort `Fresh` ensures the uniqueness of secret keys, while the argument of the sort `Name` is not strictly necessary, but it is convenient for identifying the owner of a key.

The `Decaps` algorithm is straightforwardly declared as follows:

```
op decap : Cipher PqSk -> PqKey [frozen] .
```

Unlike `Decaps`, it is a bit tricky to specify the `KeyGen` and `Encaps` procedures because they are probabilistic algorithms. For each of the two procedures, we add an argument of the sort `PqSk` to make them become deterministic procedures. With the `Encaps` procedures, we declare two separate Maude operators: `encapCipher` and `encapKey` that return the ciphertext c and the key k, respectively. We declare the following operators:

```
op pqPk        : PqSk      -> PqPk   [frozen] .
op encapCipher : PqPk PqSk -> Cipher [frozen] .
op encapKey    : PqPk PqSk -> PqKey  [frozen] .
```

The first operator models the `KeyGen` procedure, while the two others model the `Encaps` procedure. The algebraic properties of KEMs are then specified as follows:

```
op $pqKey : PqSk PqSk -> PqKey [frozen] .
eq encapKey(pqPk(S:PqSk), S2:PqSk) = $pqKey(S:PqSk, S2:PqSk) [variant] .
eq decap(encapCipher(pqPk(S:PqSk),S2:PqSk), S:PqSk)
   = $pqKey(S:PqSk, S2:PqSk) [variant] .
```

where the `variant` attribute denotes that the two equations are not regular Maude equations used for simplification, but are equations used for variant-based equational unification [12]. Here we introduce one more operator, namely `$pqKey`, which is necessary for specifying the rewritings of `encapKey` and `decap` (`Encaps` and `Decaps` steps) on proper arguments resulting in the same key. The first equation can be straightforwardly comprehended. The second equation states that given an encapsulation en and a secret key sk, a principal can perform `Decaps`(en, sk) to get the proper shared key only if en is the result of `Encaps` when taking as input the public key associated with the secret key sk.

4.2.2. ECDH and key calculation specification

To model ECDH in Maude-NPA, we first introduce the following sorts:

```
sort Scalar Point ECKey .
subsort Point < ECKey .
```

The sort `Point` represents points on the curve, which serve as ECDH public keys. The sort `Scalar` and `ECKey` represents the secret keys and shared keys, respectively. We then declare the following operators:

```
op p   : -> Point .
op sk  : Name Fresh -> Scalar [frozen] .
op gen : Point Scalar -> Point [frozen] .
op _*_ : Scalar Scalar -> Scalar [frozen assoc comm] .
```

The constant p denotes a point generator on the curve, which is publicly known by everyone including the intruder. The operator sk takes as inputs a principal name and a fresh, and outputs a scalar, which serves as a secret key. The operator gen takes as inputs a point and a (secret) scalar and returns as output another point. In particular, when the first argument is a point generator, the operator outputs a public key, which is used to send to the other peer; and when the first argument is a public key received from the opposite peer, the operator outputs a shared key. The last operator is the associative-commutative multiplication operation on scalars, thanks to the Maude attributes assoc and comm. The algebraic property of ECDH is then specified as follows:

```
eq gen(gen(P:Point, K1:Scalar), K2:Scalar)
  = gen(P:Point, K1:Scalar * K2:Scalar) [variant] .
```

Sorts PreMasterSecret and MasterSecret are introduced to represent premaster secrets and master secrets in the protocol key calculation. We then model their calculations with the following operators:

```
op pms : ECKey PqKey -> PreMasterSecret [frozen] .
op ms  : PreMasterSecret Rand Rand Point Cipher -> MasterSecret [frozen] .
```

where the sort Rand represents random numbers generated by clients and servers. A pre-master secret is the concatenation of an ECDH shared secret and a PQ KEM shared secret. Whereas, a master secret is computed by the pseudorandom function (PRF) from a pre-master secret and a seed, which is the combination of the two random numbers in the ClientHello and ServerHello messages and the ECDH public key & PQ encapsulation in the ClientKeyExchange message sent in that session.

4.2.3. Honest principal specification

We use three operators rd, sess, and cert that serve as functions to produce random numbers, session IDs, and digital certificates, respectively. We also define operators sig and enc reflecting the signature and encryption functions. All of them are as follows:

```
op  rd  : Name Fresh -> Rand [frozen] .
op sess : Name Fresh -> Session [frozen] .
op cert : Name -> Cert [frozen] .
op sig  : Name Msg -> Msg [frozen] .
op enc  : MasterSecret Msg -> Msg [frozen] .
```

With a server S, a message M, and a master-secret MS, enc(MS,M) denotes the ciphertext obtained by encrypting M by MS, while sig(S,M) denotes the signature over message M signed by the long-term private key of server S. By using a name as an argument of the signature algorithm instead of explicit private key encryption (for example, sig(priKey(S),M)), we implicitly associate a long-term private key with its owner's name. For the sake of simplicity and for reducing the size of the state space, cert(S) is used to denote the digital certificate of the server S, while in fact, the certificate must contain information about the trusted certificate authority and the public key of S. By using that simplification form, we explicitly associate a certificate with its owner's name and ignore the case when the intruder tries to fake a certificate.

Let r1, r2, and r3 be variables of the sort Fresh; C and S be variables of the sort Name; N and SS be variables of the sorts Rand and Session, respectively; PK1 and PK2 be variables of the sorts Point and PqPk, respectively. The execution of the Hybrid PQ TLS protocol up to the client's Finished message from a client's side is specified as follows:

```
:: r1,r2,r3 ::
[ nil |
 +(ch ; rd(C,r1)),
 -(sh ; N ; SS),
 -(sc ; cert(S)),
 -(ske ; PK1 ; PK2 ; sig(S, PK1 ; PK2 ; rd(C,r1) ; N)),
 +(cke ; gen(p,sk(C,r2)) ; encapCipher(PK2, pqSk(C,r3))),
 +(cf ; enc(ms(pms(gen(PK1,sk(C,r2)), encapKey(PK2, pqSk(C,r3))),
         rd(C,r1), N, gen(p,sk(C,r2)), encapCipher(PK2, pqSk(C,r3))),
    (ch ; rd(C,r1)) ++
    (sh ; N ; SS) ++
    (sc ; cert(S)) ++
    (ske ; PK1 ; PK2 ; sig(S, PK1 ; PK2 ; rd(C,r1) ; N)) ++
    (cke ; gen(p,sk(C,r2)) ; encapCipher(PK2, pqSk(C,r3))))
  ),
nil ]
```

where ch, sh, sc, ske, cke, and cf are constants of the sort Msg, which are acronyms of ClientHello, ServerHello, Server Certificate, ServerKeyExchange, ClientKeyExchange, and client's Finished. The ++ operator denotes message concatenation. The strand says that the client starts a new connection by sending a ClientHello message with a random number denoted by rd(C,r1). When the client receives back a ServerHello message, a valid server Certificate message, and a valid ServerKeyExchange message with ECDH and PQ public keys by means of PK1 and PK2, the client will send a ClientKeyExchange message with the client's ECDH public key exchange and the encapsulation computed with PK2 as one of the inputs. Together with that message, the client also sends a Finished message, whose content is derived by encrypting the concatenation of all messages exchanged so far (denoted by ++) under the master secret key. Note that for the sake of simplicity and for reducing the size of the state space, here we use the master secret as the symmetric key for encryption of the Finished message, but actually in the protocol design, such a symmetric key is computed by the PRF function from the master secret and the two random numbers in the ClientHello and ServerHello messages. In addition, we also suppose that client authentication is not requested, we eliminate ServerHelloDone & ChangeCipherSpec messages, and some parameters in the Hello messages such as protocol version, cipher suites, and PQ KEMs parameters are excluded. Note also that the use of ++ is unnecessary. For example, based on the definition of _++_, (ch ; rd(C,r1)) ++ (sh ; N ; SS) is rewritten to (ch ; rd(C,r1) ; sh ; N ; SS), so we can exclude the use of ++. However, we define and keep on using it because we want to show each message separately so that readers can easily follow.

In a similar way, we specify the protocol execution from a server's side up to the client's Finished message by the following strand:

```
:: r1,r2,r3,r4 ::
[ nil |
```

```
 -(ch ; N),
 +(sh ; rd(S,r1) ; sess(S,r2)),
 +(sc ; cert(S)),
 +(ske ; gen(p,sk(S,r3)) ; pqPk(pqSk(S,r4)) ;
     sig(S, gen(p,sk(S,r3)) ; pqPk(pqSk(S,r4)) ; N ; rd(S,r1))),
 -(cke ; PK1 ; CP),
 -(cf ; enc(ms(pms(gen(PK1,sk(S,r3)), decap(CP, pqSk(S,r4))),
           N, rd(S,r1), PK1, CP),
        (ch ; N) ++
        (sh ; rd(S,r1) ; sess(S,r2)) ++
        (sc ; cert(S)) ++
        (ske ; gen(p,sk(S,r3)) ; pqPk(pqSk(S,r4)) ;
           sig(S, gen(p,sk(S,r3)) ; pqPk(pqSk(S,r4)) ; N ; rd(S,r1))) ++
        (cke ; PK1 ; CP))
  ),
nil ]
```

where `CP` is a variable of the sort `Cipher`. Note that in both strands, we exclude the appearance of the server's Finished message because they are too long to show all. Readers can find the complete specification from the webpage provided in Section 1.

4.2.4. Intruder capabilities

Maude-NPA uses the standard Dolev-Yao intruder model [6], that is the intruder can, for example, intercept and glean information from any message in the network; fake, synthesize, and send messages based on the gleaned information. Similar to the capability in concatenation messages presented in Section 4.1, we specify the deconcatenation message capability for the intruder as follows:

```
:: nil :: [ nil | -(M1 ; M2), +(M1), nil ] &
:: nil :: [ nil | -(M1 ; M2), +(M2), nil ] &
```

The intruder can generate by himself/herself any random number and any point serving as an ECDH private key exchange:

```
:: r :: [ nil | +(rd(i,r)), nil ] &
:: r :: [ nil | +(sk(i,r)), nil ] &
```

Note that these two strands consume some fresh `r`.

For KEMs, if there are some public key `PK2`, secret key `SK`, and encapsulation `CP` that are available to the intruder, then the intruder can derive some appropriate encapsulations and keys as follows:

```
:: nil :: [ nil | -(PK2), -(SK), +(encapCipher(PK2,SK)), nil ] &
:: nil :: [ nil | -(PK2), -(SK), +(encapKey(PK2,SK)), nil ] &
:: nil :: [ nil | -(CP), -(SK), +(decap(CP,SK)), nil ] &
```

With ECDH, an important assumption we suppose is that the intruder can break the key exchange security by utilizing the power of quantum computation. That is, if the intruder knows the two ECDH public keys exchanged between a client and a server, then the intruder can derive the shared secret key. This is done by the following strand:

```
:: nil :: [ nil | -(gen(p,K1)), -(gen(p,K2)), +(gen(p,K1 * K2)), nil ]
```

where `K1` and `K2` are variables of `Scalar`.

There are also some more strands specifying the intruder capabilities, but we omit to present all of them here. Again, readers can find them from the webpage presented in Section 1.

5. Related work

Hülsing et al. [14] have verified the security of their proposed post-quantum WireGuard (PQ-WireGuard) protocol [14]. This is a quantum-resistant version of the WireGuard protocol [15], which is a lightweight and high-performance VPN protocol. The verification confirms that the protocol enjoys the desired security properties inherited from the WireGuard protocol and also resists attacks using a large-scale quantum computer. The verification used the Tamarin prover [8], which is known as one of the state-of-the-art formal verification tools for symbolic analysis of cryptographic protocols. Similar to Maude-NPA, they have first symbolically modeled the primitives, messages, etc. used in the protocol as function symbols and terms, and then specified the desired security properties. However, unlike Maude-NPA, several commonly used primitives are pre-defined as built-in functions in Tamarin, such as the Diffie-Hellman key exchange algorithm, symmetric and asymmetric encryption, hashing, and digital signatures, while Maude-NPA leaves all of these definitions to human users. To prove the protocol enjoys the desired properties, they also introduced some auxiliary lemmas, which are also needed to prove. Conjecturing lemmas, however, is one of the most intellectual tasks in formal verification, which is not a new issue in the theorem proving field.

Using some additional lemmas to complete verification in Tamarin is called the interactive mode, distinguishing it from the fully automated mode. This way of verification is useful when the tool fails to prove properties in the fully automated mode or the time taken is too long. Tamarin operates based on multiset rewriting and its verification algorithm is based on constraint solving. Similar to Maude-NPA, the tool can handle an unbounded number of sessions (executions) of protocols. Once the tool terminates, it returns either proof of security correctness or an attack. A Tamarin specification is essentially a state machine where each state is a multiset of *facts*. Transitions between states are defined by *rules*. Rules specify the protocol execution, the behavior of honest parties as well as the capabilities of the intruder. A security property is modeled as a trace property, and then Tamarin checks the satisfiability and/or the validity of the property. If it is the validity checking, Tamarin first converts it to checking the satisfiability of the negated formula formalizing the property. Constraint solving is then used to perform an exhaustive, symbolic search for executions with the trace until a satisfying trace is found or no more rewrite rules can be applied. Roughly speaking, the negation of the formula formalizing the validity property in Tamarin corresponds to the attack pattern in Maude-NPA, and the satisfying trace, if found, in Tamarin corresponds to the initial state, if found, in Maude-NPA.

Cremers et al. [16] have presented a comprehensive security analysis of TLS 1.3 [17], precisely, the TLS 1.3 draft 21 release candidate. They used the Tamarin tool to verify the claimed security requirements, which were stated in the draft, with respect to a Dolev-Yao intruder. Their analysis considers all the possible interactions of the available handshake modes in TLS 1.3,

such as pre-shared key (PSK) based resumption and zero round trip time (0-RTT), which are new mechanisms only available from version 1.3. Similar to the work in [14], to complete the security verification, they have conjectured a number of auxiliary lemmas, with some manual interaction in the Tamarin interactive mode.

6. Conclusion

This paper has presented a formal specification of the Hybrid PQ TLS protocol in Maude-NPA, which is the first step toward conducting a security analysis of the protocol with Maude-NPA. In the next step, from the formal specification, we are going to specify the attack states and execute experiments to check the satisfiability of the secrecy property and the authentication property. In addition to the original Maude-NPA, we are also going to use a parallel version of Maude-NPA [11] for the experiments to make the best use of multicore architectures, for which we strongly believe that a better running performance than the original one in terms of verification time will be obtained.

To prepare for the quantum computing era, which may become in a near future, security analysis by formal methods is necessary to verify and construct secure post-quantum cryptosystems. Security verification by Maude-NPA is fully automated, that is no manual effort is required once the formal specification and the attack pattern are provided, but it may take a quite long time if the state space of the system under verification is huge (i.e., state explosion). Therefore, regarding the long-term future work, we are also interested in formal verifications of post-quantum cryptographic protocols using some interactive approaches that take less time but often require some manual human user effort. CafeOBJ/proof score approach [18, 19] is a promising way, for which we plan to use it to tackle the formal verification of the Hybrid PQ TLS protocol.

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] 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.

[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] R. Azarderakhsh, M. Campagna, C. Costello, L. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, et al., Supersingular isogeny key encapsulation (2020). URL: https://sike.org/files/SIDH-spec.pdf.

[5] N. Aragon, P. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Güneysu, C. A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, G. Zémor,

Bike: Bit flipping key encapsulation - round 3 submission, 2019. URL: https://bikesuite.org/files/v4.2/BIKE_Spec.2021.09.29.1.pdf.

[6] 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.

[7] F. J. Thayer, J. C. Herzog, J. D. Guttman, Strand spaces: Why is a security protocol correct?, in: Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998, Proceedings, IEEE Computer Society, 1998, pp. 160–171. doi:10.1109/SECPRI.1998.674832.

[8] S. Meier, B. Schmidt, C. Cremers, D. Basin, The TAMARIN Prover for the Symbolic Analysis of Security Protocol, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 696–701. doi:10.1007/978-3-642-39799-8_48.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott (Eds.), All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, volume 4350 of *Lecture Notes in Computer Science*, Springer, 2007. doi:10.1007/978-3-540-71999-1.

[10] J. Meseguer, Twenty years of rewriting logic, in: P. C. Ölveczky (Ed.), Rewriting Logic and Its Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 15–17.

[11] C. M. Do, A. Riesco, S. Escobar, K. Ogata, Parallel Maude-NPA for cryptographic protocol analysis, in: K. Bae (Ed.), Rewriting Logic and Its Applications - 14th International Workshop, WRLA@ETAPS 2022, Munich, Germany, April 2-3, 2022, Revised Selected Papers, volume 13252 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 253–273. doi:10.1007/978-3-031-12441-9_13.

[12] S. Escobar, R. Sasse, J. Meseguer, Folding variant narrowing and optimal variant termination, in: P. C. Ölveczky (Ed.), Rewriting Logic and Its Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 52–68.

[13] R. M. Needham, M. D. Schroeder, Using Encryption for Authentication in Large Networks of Computers, Commun. ACM 21 (1978) 993–999. doi:10.1145/359657.359659.

[14] A. Hülsing, K. Ning, P. Schwabe, F. Weber, P. R. Zimmermann, Post-quantum WireGuard, in: 2021 IEEE Symposium on Security and Privacy, 2021, pp. 304–321. doi:10.1109/SP40001.2021.00030.

[15] J. A. Donenfeld, WireGuard: Next generation kernel network tunnel, in: 24th Annual Network and Distributed System Security Symposium, NDSS 2017, 2017.

[16] C. Cremers, M. Horvat, J. Hoyland, S. Scott, T. van der Merwe, A comprehensive symbolic analysis of TLS 1.3, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017, pp. 1773–1788. doi:10.1145/3133956.3134063.

[17] E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.3, RFC 8446, 2018. doi:10.17487/RFC8446.

[18] K. Ogata, K. Futatsugi, Compositionally Writing Proof Scores of Invariants in the OTS/CafeOBJ Method, J. Univers. Comput. Sci. 19 (2013) 771–804. doi:10.3217/jucs-019-06-0771.

[19] K. Ogata, K. Futatsugi, Proof scores in the OTS/CafeOBJ method, in: FMOODS 2003, 2003, pp. 170–184. doi:10.1007/978-3-540-39958-2_12.