UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Informatics

Formal verification of Hybrid Post-Quantum TLS protocol

End of Degree Project

Bachelor's Degree in Informatics Engineering

AUTHOR: Sánchez Marco, Adrián

Tutor: Escobar Román, Santiago

ACADEMIC YEAR: 2023/2024

Resum

Amb l'arribada de la computació quàntica, garantir la seguretat dels protocols de comunicació s'ha tornat encara més crucial. Aquest estudi explora la verificació formal d'un protocol de Seguretat de la Capa de Transport dissenyat per a resistir les amenaces quàntiques. Mentre que els mètodes criptogràfics tradicionals són efectius contra adversaris de la computació clàssica, són susceptibles a atacs potenciats per la computació quàntica. Per tant, hi ha una necessitat real de transicionar a la criptografia post-quàntica, que ofereix resistència contra amenaces futures.

Aquesta investigació es centra en verificar formalment un protocol TLS post-quàntic per a proporcionar una garantia matemàtica de les seues propietats de seguretat. La verificació formal ofereix un mètode rigorós per analitzar la correcció i robustesa dels protocols criptogràfics, reduint el risc de vulnerabilitats no detectades. En emprar mètodes formals, examinem el disseny del protocol, la seua implementació i primitives criptogràfiques per a identificar possibles debilitats.

La verificació formal del protocol TLS post-quàntic no només millora la nostra comprensió de les seues garanties de seguretat, sinó que també subratlla la importància d'una validació rigorosa en el disseny criptogràfic.

Paraules clau: Maude-NPA, TLS, criptografia, post-quàntic

Resumen

Con la llegada de la computación cuántica, garantizar la seguridad de los protocolos de comunicación se ha vuelto aún más crucial. Este estudio explora la verificación formal de un protocolo de Seguridad de la Capa de Transporte diseñado para resistir las amenazas cuánticas. Mientras que los métodos criptográficos tradicionales son efectivos contra adversarios de la computación clásica, son susceptibles a ataques potenciados por la computación cuántica. Por lo tanto, hay una necesidad real de transicionar a la criptografía post-cuántica, que ofrece resistencia contra amenazas futuras.

Esta investigación se centra en verificar formalmente un protocolo TLS post-cuántico para proporcionar una garantía matemática de sus propiedades de seguridad. La verificación formal ofrece un método riguroso para analizar la corrección y robustez de los protocolos criptográficos, reduciendo el riesgo de vulnerabilidades no detectadas. Al emplear métodos formales, examinamos el diseño del protocolo, su implementación y primitivas criptográficas para identificar posibles debilidades.

La verificación formal del protocolo TLS post-cuántico no solo mejora nuestra comprensión de sus garantías de seguridad, sino que también subraya la importancia de una validación rigurosa en el diseño criptográfico.

Palabras clave: Maude-NPA, TLS, criptografía, post-cuántico

Abstract

With the arrival of quantum computing, ensuring the security of communications protocol has become even more crucial. This study explores the formal verification of a Transport Layer Security protocol designed to withstand quantum menaces. While traditional cryptographic methods are effective against classical computing adversaries, they are susceptible to attacks empowered by quantum computation. Hence, there is a real need to transition to post-quantum cryptography, which offers resilience against future threats.

This research focuses on formally veryfiying a post-quantum TLS protocol to provide mathematical assurance of its security properties. Formal verification offers a rigorous method for analyzing the correctness and robustness of cryptographic protocols, reducing the risk of undetected vulnerabilities. By employing formal methods, we scrutinize the protocol's design, implementation, and cryptographic primitives to identify a potential weaknesses.

The formal verification of the post-quantum TLS protocol not only enhances our understanding of its security assurances but also underscores the significance of rigorous validation in cryptographic design.

Key words: Maude-NPA, TLS, cryptography, post-quantum

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In the realm of cybersecurity, the ability to protect data integrity and confidentiality during transmission has always been crucial. As digital communication technologies advance, so also do the methods employed by attackers to compromise these systems. This dynamic environment has driven the need for continuous improvements in security protocols to safeguard sensitive information. The emergence of quantum computing represents a significant paradigm shift, threatening the foundations of current cryptographic systems. In response to these developments, the focus has increasingly shifted towards designing and implementing new protocols that can withstand the capabilities of quantum machines. This thesis addresses these challenges by exploring the Hybrid Post-Quantum Transport Layer Security protocol, aiming to enhance data protection in an era where quantum computing poses a tangible threat.

1.1 Motivation

In the ever-evolving landscape of cybersecurity, the integrity and confidentiality of data transmissions are paramount. As technological advancements push the boundaries of digital communication, traditional encryption protocols face increasing scrutiny, particularly with the advent of quantum computing. The potential for quantum computers to break widely used encryption schemes necessitates the development of new, resilient security protocols. This thesis aims to address this critical need by focusing on Hybrid Post-Quantum Transport Layer Security. By contributing to the refinement and enhancement of cybersecurity measures through robust protocol modeling, this research endeavors to fortify data protection against emerging quantum threats. The motivation behind this thesis is to advance the field of cybersecurity by providing innovative solutions that enhance the resilience of data communication systems in a post-quantum world.

1.2 Objectives

The primary objective of this thesis is identifying and addressing potential attacks, to rigorously analyze the protocol for possible attack vectors and vulnerabilities. This involves employing advanced modeling techniques to simulate various attack scenarios, thereby identifying weaknesses. By achieving this objective, the research will contribute valuable insights and solutions to the cybersecurity domain, aiding in the transition to secure post-quantum communication standards.

1.3 Structure of the Thesis

The structure of this thesis is organized as follows:

1. **Introduction**: this chapter outlines the motivation for the research, specifies the objectives, and provides an overview of the thesis structure.
2. **State of the Art**: this section reviews current advancements and existing research related to TLS and post-quantum cryptography, setting the context for the proposed protocol.
3. **Maude NPA**: the formal language and tool Maude NPA, used for protocol modeling, are introduced in this section. The relevance and application of Maude NPA in the context of this research are explained.
4. **Transport Layer Security protocol**: a foundational overview of the Transport Layer Security (TLS) protocol is provided in this chapter, including its core functions and significance in secure communications.
5. **Hybrid Post-Quantum Transport Layer Security protocol**: this section delves into the specifics of the Hybrid Post-Quantum TLS protocol, detailing its design, functionality, and the rationale behind its development.
6. **Protocol Modeling and Results**: the results of the protocol modeling process are presented in this chapter, including identified vulnerabilities and proposed solutions. The effectiveness of the Hybrid Post-Quantum TLS protocol in mitigating potential attacks is evaluated.
7. **Conclusion**: the final conclusions of what has been achieved in the thesis.
8. **Bibliography**: this section includes all references cited throughout the thesis.
9. **Sustainable Development Goals**: supplementary material, including details of the Specification Development Guidelines (SDG) relevant to the protocol modeling, is provided in this section.

CHAPTER 2

State of the Art

The field of cybersecurity has undergone significant transformations as new technologies and threats emerge. This chapter provides a comprehensive review of the current state of research and development in the areas pertinent to this thesis: Transport Layer Security (TLS) and post-quantum cryptography.

2.1 Transport Layer Security (TLS)

Transport Layer Security (TLS) is a widely adopted protocol designed to provide secure communication over a computer network. It is employed extensively to secure connections between clients and servers on the Internet, ensuring data integrity, confidentiality, and authentication.

2.1.1. History and Evolution

TLS evolved from the Secure Sockets Layer (SSL) protocol developed by Netscape in the 1990s. SSL 3.0, introduced in 1996, was succeeded by TLS 1.0 in 1999. Since then, several versions of TLS have been released, including TLS 1.1 (2006), TLS 1.2 (2008), and TLS 1.3 (2018). Each version has introduced improvements in security and performance, addressing vulnerabilities discovered in previous iterations.

2.1.2. Core Components

TLS operates through a combination of cryptographic algorithms to secure data during transmission. The core components of TLS include:

- **Handshake Protocol**: establishes a secure connection between the client and server, negotiating cryptographic parameters and exchanging keys.
- **Record Protocol**: handles the encryption and decryption of data transmitted over the network.
- **Alert Protocol**: provides a mechanism for signaling errors and connection issues.
- **Change Cipher Spec Protocol**: updates the cipher settings during the session.

2.1.3. Current Challenges

Despite its widespread use, TLS faces several challenges, including susceptibility to certain types of attacks such as man-in-the-middle (MITM) and downgrade attacks. Efforts to address these challenges have led to the development of more secure versions and improved configurations.

2.2 Post-Quantum Cryptography

The advent of quantum computing poses a significant threat to classical cryptographic systems, including those employed by TLS. Post-quantum cryptography refers to cryptographic algorithms designed to be secure against the capabilities of quantum computers without requiring quantum cryptography.

2.2.1. Quantum Threats to Classical Cryptography

Quantum computers leverage principles of quantum mechanics to perform computations that are infeasible for classical computers. Algorithms such as Shor's algorithm [12] could potentially break widely used encryption schemes like RSA and ECC by efficiently solving problems that are currently intractable for classical computers.

2.2.2. NIST Post-Quantum Cryptography Standardization

The National Institute of Standards and Technology (NIST[1]) has been actively involved in the standardization of post-quantum cryptographic algorithms. The NIST Post-Quantum Cryptography Project aims to evaluate and standardize quantum-resistant cryptographic algorithms, providing guidelines for their implementation and deployment.

2.2.3. Hybrid Post-Quantum TLS

Given the transitional nature of current cryptographic systems, hybrid approaches that combine classical and post-quantum cryptographic techniques are being explored. Hybrid cryptography aims to leverage the strengths of both classical and quantum-resistant algorithms to enhance security during the transition period. In this context, Hybrid Post-Quantum TLS protocol [4] integrates post-quantum cryptographic algorithms into the existing TLS framework. This integration seeks to provide enhanced security against quantum attacks while maintaining compatibility with current systems. Research in this area focuses on optimizing performance and ensuring robust security properties in the face of evolving threats.

2.3 Conclusion

The state of the art in TLS and post-quantum cryptography reflects ongoing efforts to address the challenges of securing digital communication in a rapidly changing technological landscape. The development of hybrid protocols represents a promising direction for enhancing the resilience of security systems against future quantum threats. The following chapters will delve into the specifics of Maude NPA, TLS, and Hybrid Post-Quantum TLS, building on the insights gained from this review.

[1]NIST: `https://www.nist.gov/`

CHAPTER 3
Maude-NPA

The Maude-NRL Protocol Analyzer (Maude-NPA [1]) is a tool based on Maude [2], a modelling, analysis and programming language focused on formal specifications using algebraic terms. The main objective of Maude-NPA is centered around verifying cryptographic communication protocols, particularly their properties. This tool includes algebraic properties that are not present in other tools, such as encryption and decryption cancellation, Abelian groups (including XOR), exponentiation, and homomorphic encryption.

Maude-NPA utilizes a similar approach to the original NRL Protocol Analyzer, based on unification. It conducts a backward search from the final state to ensure neither whether a state is reachable nor not. Far from the original NPA, it incorporates theoretical foundations in rewrite logic and narrowing, in addition to providing support for a broader range of equational theories, embracing commutativity, along with associativity-commutativity and associativity-commutativity-identity.

3.1 Protocol Modeling

When specifying a cryptographic protocol in Maude-NPA [1], we will use a file with the .maude extension (e.g., protocol.maude). Once we start with the modeling, we should develop three modules within the program, namely the syntax module, the algebraic module, and the behavior module.

3.1.1. Syntax Module

The syntax module is the starting point of defining a protocol's syntax, which includes the signatures of *sorts* and *operators*. A sort is defined by utilizing the *sort* keyword, succeeded by its identifier (the sort name) and a period, for example:

```
sort ⟨sort_1⟩ .
```

In order to declare multiple sorts, instead of using the standard declaration method, the *sorts* keyword will be used instead:

```
sorts ⟨sort_1⟩ ... ⟨sort_k⟩ .
```

Subsort relationships, which indicate subtype hierarchies among sorts, are established using the *subsort* keyword:

```
subsort ⟨sort_1⟩ < ⟨sort_2⟩
```

This indicates that the first sort is considered to be a subtype of the second, implying that the first sort inherits the properties of the second sort, thereby establishing a hierarchical relationship between the two sorts.

Operators are declared using the keyword *op*. There are four components to declaring an operator: the operator name, the list of sort arguments, which is preceded by a colon, the resulting sort (*range sort*), preceded by ->, and the operator attributes, found at the end and enclosed in brackets:

```
op ⟨op_name⟩ : ⟨sort_1⟩ ... ⟨sort_k⟩ -> ⟨sort⟩ [⟨op_attributes⟩] .
```

Furthermore, in Maude-NPA, special sorts are utilized. These especial sorts are imported using the following code:

```
protecting DEFINITION-PROTOCOL-RULES .
```

This code imports the special types, which are essential for defining various elements within the system. The special types included are as follows:

- **Msg:** user-defined sorts can be subtypes of Msg. No user-defined sort can be a supertype of it. This sort cannot be empty; it's imperative to define at least one symbol of the Msg sort or a subtype of Msg.

- **Fresh:** the Fresh sort is employed to designate terms that must be unique. It's commonly applied as an attribute to data that requires uniqueness, such as a nonce or a session key, for instance, "nonce(A,r)" or "key(A,B,r)" where r is a variable of the Fresh sort. Only variables can possess the Fresh sort; constants or function symbols cannot.

- **Public:** the Public sort designates terms that are publicly accessible, hence assumed to be known by any potential intruder. Being a subtype of Msg, explicitly declaring "subsort Public < Msg" is unnecessary. This sort cannot be left empty.

The following code depicts an example of the syntactic module. In the code, dashes are used to comment the code by employing a sequence of three consecutive dashes:

```
--- Syntax module
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  --- Importing sorts Msg, Fresh, Public
  protecting DEFINITION-PROTOCOL-RULES .

  sorts Name Nonce Secret .
  subsort Name < Msg .

  op secret_operator : Name Fresh -> Secret [frozen] .
endfm
```

3.1.2. Algebraic Module

The second module specifies the algebraic properties of the operators. It is made up of unconditional equations that are defined by the keyword *eq*, featuring a term (left side), followed by the equals sign, another term (right side), and optionally enclosed within square brackets, a list of statement attributes. This declaration concludes with white a space and a period. Hence, the typical format is as follows:

```
eq ⟨term_1⟩ = ⟨term_2⟩ [⟨statement_attributes⟩] .
```

In the context of protocol development, it is crucial to tailor the algebraic properties to suit the specific requirements of the protocol. This can be achieved by overwriting the module with the desired algebraic properties. One approach is to utilize variant equations exclusively, following the syntax:

```
eq Lhs = Rhs [variant] .
```

Alternatively, dedicated unification algorithms can be employed by utilizing equations tailored for this purpose. It's important to note that the *owise* attribute cannot be utilized in this context, emphasizing the necessity of adherence to protocol-specific guidelines and constraints.

The provided code exemplifies the algebraic module. Comments within the code are denoted by a sequence of three consecutive dashes:

```
--- Algebraic module
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .

  eq h(K:Key,
       e(key:Key,
         msg:Msg)) =
     msg:Msg [variant] .
  eq x(K:Key,
       d(key:Key,
         msg:Msg)) =
     msg:Msg [variant] .
  eq exponentiation(exponentiation(p:Point,
                                   sc1:Scalar),
                    sc2:Scalar) =
     exponentiation(p:Point,
                    sc1:Scalar * sc2:Scalar) [variant] .
endfm
```

3.1.3. Behavior Module

The third module delineates the practical operations of the protocol through a strand-theoretic notation. Within this module, there are descriptions of the capabilities of the *Dolev-Yao* intruder, alongside conventional strands or processes representing the actions of principals. Additionally, it encompasses attack states, which outline behaviors that we aim to demonstrate as impossible.

The provided code serves as an illustration of the behavior module, demonstrating how specific functionalities and rules are implemented within the system:

```
fmod PROTOCOL-SPECIFICATION is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .
  protecting DEFINITION-PROTOCOL-RULES .
  protecting DEFINITION-CONSTRAINTS-INPUT .

  eq STRANDS-DOLEVYAO =
    :: nil :: [nil | -(M1 ; M2), +(M1), nil]
    [nonexec] .

  eq STRANDS-PROTOCOL =
    :: r1, r2 :: [nil | +(A ; exp(g, n(A, ra)))]
    [nonexec] .

  eq ATTACK-STATE(0) =
    :: r :: [nil, +(pk(a, N ; n(b,r))) | nil]
    [nonexec] .
endfm
```

3.2 Protocol Strands

Protocols form the backbone of communication in computer networks, facilitating the exchange of data between various entities. However, this very ubiquity makes them susceptible to exploitation by malicious actors seeking to compromise network integrity or glean sensitive information. Understanding the behavior of protocols under different scenarios is crucial for identifying and mitigating potential security risks. Protocol strands emerge as a valuable concept in this domain, providing a systematic framework for representing protocol behavior across different dimensions.

At its core, a protocol strand encapsulates the behavioral aspects of a protocol, offering insights into how it operates under varying conditions. There are three primary components that constitute a protocol strand: Attacker Skills, Client/Server Behavior and Attack Patterns Strands.

3.2.1. Attacker Skills Strand

The first component, `STRANDS-DOLEVYAO`, focuses on delineating the capabilities and tactics employed by potential attackers. This includes understanding their proficiency in exploiting vulnerabilities, executing sophisticated attacks, and evading detection mechanisms. By encoding attacker skills within the protocol strand, analysts gain a deeper understanding of the threat landscape and can tailor defensive strategies accordingly.

In order to specify the abilities that the intruder can perform, we will use the *Dolev-Yao* rules. The Dolev-Yao model provides a formal framework for analyzing the security of cryptographic protocols. At its core, the model assumes an adversarial environment where intruders have unrestricted access to communication channels and possess omnipotent capabilities to intercept, manipulate, and inject messages. The Dolev-Yao rules encapsulate various actions that an intruder can perform based on the structure and semantics of the protocol being analyzed. These actions include message interception, message modification, message fabrication, and message replay, among others.

Below is a snippet of code representing the possible abilities an attacker can perform, showcasing the various actions and strategies that an attacker might employ:

```
eq STRANDS-DOLEVYAO =
  :: nil :: [nil | -(M1 ; M2),
                   +(M1), nil] &
  :: nil :: [nil | -(M1 ; M2),
                   +(M2), nil]
  [nonexec] .
```

In the first line, the intruder intercepts a message sequence (M1 followed by M2), and then sends message M1, potentially disrupting the normal flow of communication. In the second line, the intruder again intercepts the message sequence, but this time sends message M2 instead, potentially aiming to manipulate the communication or exploit vulnerabilities in the protocol. These lines outline possible attack scenarios within the protocol model, illustrating the intruder's capabilities and intentions.

3.2.2. Client/Server Strand

The second component, STRANDS-PROTOCOL, delves into elucidating the operational intricacies and functional dynamics inherent in the protocol's design. It elucidates how the protocol operates under normal conditions, detailing the sequence of actions and interactions among participating entities to achieve predefined objectives. By articulating the protocol's behavior within this module, analysts gain a nuanced understanding of its operational logic, error handling mechanisms, and state transitions.

This strand primarily focuses on modeling the protocol's operational behavior capturing the sequence of actions performed by legitimate entities and the corresponding system responses. It delineates the expected behavior of entities participating in the protocol, encompassing message exchanges, state transitions, and protocol-specific operations.

To specify the operational behavior of the protocol, we employ a formal modeling approach that abstracts the system's operational logic into a set of rules and constraints. These rules define the permissible actions and state transitions within the protocol, encapsulating the protocol's operational semantics and functional requirements.

Below is a snippet of code representing the operational behavior of the protocol, where + represents the sending of messages and - the reception of two principals interactions between them graphically represented in Figure 3.1:

```
eq STRANDS-PROTOCOL =
  :: r1 ::
  [nil | +(client_hello ; random(Client, r1)),
         -(server_hello ; Nonce ; Session)] &
  :: r1, r2 ::
  [nil | -(client_hello ; Nonce),
         +(server_hello ; random(Server, r1) ; session(Server, r2))]
  [nonexec] .
```

The provided code delineates the operational sequence of the first step of the Transport Layer Security Protocol exchange. The client starts the communication by sending client_hello message along with a randomly generated value. The server catches the messages where we represent the random value generated as a Nonce.

After the server receives the message with the Nonce, it responds by sending the server_hello message to the client. In this message, the server generates another random value along with initiating a session. This server_hello serves as a confirmation of the connection establishment and sets the foundation for further communication between the server and the client within the session. The client catches the messages representing the random value generated as a Nonce and the Session.

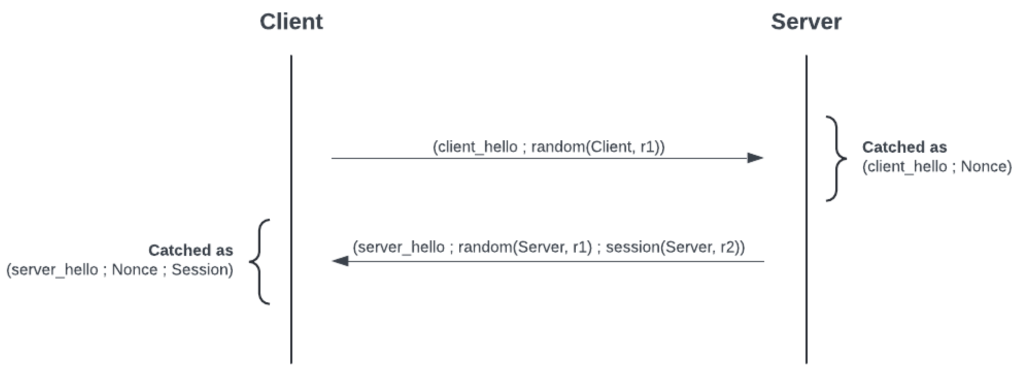

Client
Server
(client_hello ; random(Client, r1))
Catched as
(client_hello ; Nonce)
(server_hello ; random(Server, r1) ; session(Server, r2))
Catched as
(server_hello ; Nonce ; Session)

Figure 3.1: Message exchange sequence diagram

3.2.3. Attack Patterns

The last component, the Attack Patterns, denoted by ATTACK-STATE(n), focuses on defining the final states utilized for backward search in the protocol modeling language. This segment is crucial for identifying potential vulnerabilities and understanding the ramifications of successful attacks.

In specifying an attack state within the Attack Patterns, users can only define the first two sections, which include:

- **Set of Strands:** this section outlines the strands expected to manifest in the attack, indicating the specific components targeted by the intruder.
- **Intruder Knowledge:** describes the information and capabilities possessed by the intruder, shaping the tactics and strategies employed in the attack.
- **Never Patterns**: specifies a combination of states that cannot be achieved.

For instance, in the context of NSPK (Needham-Schroeder Public Key), a standard attack entails the intruder acquiring knowledge of the nonce generated by Bob. Therefore, to depict this scenario within an attack pattern, it is imperative to include a completed execution of Bob's strand, denoted by the vertical bar "|" at the conclusion, thereby specifying the nonce **n(b,r)**.

A representation of the Attack Patterns for the NSPK protocol might resemble the following:

```
eq ATTACK-STATE(0) =
  :: r ::
  [nil, -(pk(b,a ; N)),
        +(pk(a, N ; n(b,r))),
        -(pk(b,n(b,r))) | nil]
  || n(b,r) inI
  || nil
  || nil
  || nil
  [nonexec] .
```

This depiction encapsulates the critical elements necessary to describe an attack scenario within the NSPK protocol, providing insights into potential security vulnerabilities and informing defensive strategies.

3.3 Attack-search Commands

Maude-NPA provides a range of commands to search for potential attack scenarios. Users can execute commands like: **run**, **digest**, **summary**, **initials**, **debug**, and **ids** by typing **red** followed by the command name and a period. Each command serves a distinct purpose in analyzing the protocol's security state. To utilize these commands, it is necessary to specify the attack state being searched for and the number of backward reachability steps to compute.

Run Command: the **run** command is fundamental in Maude-NPA, providing the states at the frontier of the search tree for a given depth. For instance, the command:

```
Maude> red run(0, 4) .
```

instructs Maude-NPA to build the backwards reachability tree up to a depth of 4 for the attack state identified by the number 0 and to return the leaves of that tree. Additionally, any state at depth 4 without children will also be returned, as it is considered a leaf of the search tree.

Digest Command: for users interested only in the active information from the states without all the details, the **digest** command can be used. This command excludes the strands and negative facts in the intruder's knowledge.

```
Maude> red digest(0, 4) .
```

Summary Command: if the current states of the reachability tree are not a concern, the **summary** command can be used. This command provides only the count of states discovered in the leaves of the reachability tree and specifies how many of those are initial states, meaning they are potential solutions to the attack.

```
Maude> red summary(0, 4) .
```

Initials Command: an alternative to the **run** command is the **initials** command, which outputs only the initial states instead of all the leaves.

```
Maude> red initials(0, 4) .
```

Debug Command: in Maude-NPA, states contain additional internal data that can be revealed using the **debug** command. This might include details such as never patterns linked to each state or internal data regarding various optimizations.

```
Maude> red debug(0, 4) .
```

Ids Command: when the information provided by the **run** or **digest** commands is not needed, the **ids** command can be used to display only the identifiers of all states discovered in the leaf nodes of the reachability tree.

```
Maude> red ids(0, 4) .
```

CHAPTER 4

Transport Layer Security protocol

Transport Layer Security 1.2 (TLS 1.2) [3] serves as a fundamental security protocol extensively utilized to ensure data privacy and protection in internet communications. A primary application of TLS involves encrypting interactions between web applications and servers, such as the connections established when web browsers access websites. Beyond web traffic, TLS also secures other forms of communication, such as e-mail, instant messaging, and Voice over IP (VoIP).

TLS functions by creating a secure connection, depicted in Figure 4.1, between two endpoints, effectively safeguarding against eavesdropping and tampering. This is accomplished using a mix of cryptographic methods: symmetric encryption to ensure data confidentiality, asymmetric encryption to facilitate secure key exchange, and hashing to verify data integrity. This multifaceted strategy guarantees that transmitted data remains confidential and intact, thus preserving the integrity and security of communications over the internet.

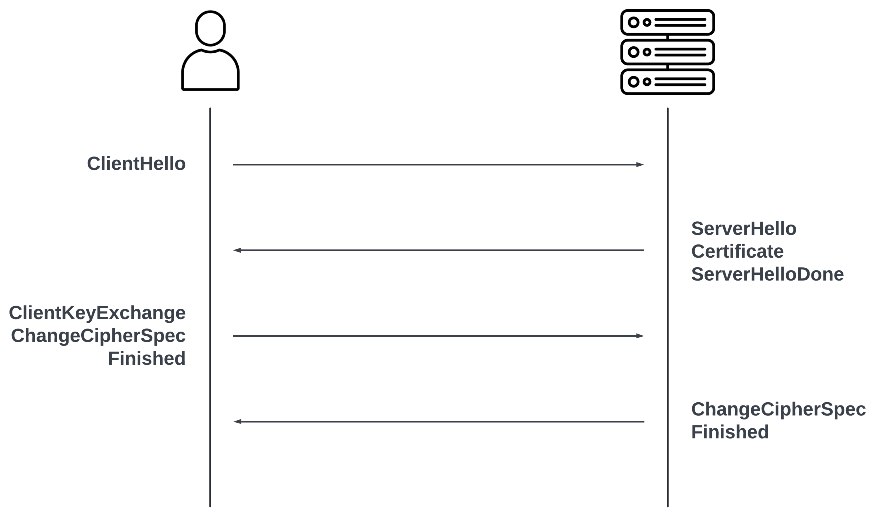


Figure 4.1: TLS protocol

4.1 TLS Handshake protocol

The Handshake Protocol operates as a higher-level client of the TLS Record Protocol. It facilitates the negotiation of secure session attributes. Handshake messages are provided to the TLS record layer, where they are enclosed in one or multiple TLSPlaintext structures, shown in Listing 4.1.

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

Listing 4.1: TLSPlaintext Struct

We can find two variables of the type `ContentType` and `ProtocolVersion`, which define the type of content being transmitted and the version of the protocol used. This can be seen in Listing 4.2.

```
struct {
    uint8 major;
    uint8 minor;
} ProtocolVersion;

enum {
    change_cipher_spec(20), alert(21), handshake(22),
    application_data(23), (255)
} ContentType;
```

Listing 4.2: ProtocolVersion Struct and ContentType Enum

These structures are then handled and sent based on the current active session state.

```
struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (HandshakeType) {
        case hello_request:       HelloRequest;
        case client_hello:        ClientHello;
        case server_hello:        ServerHello;
        case certificate:         Certificate;
        case server_key_exchange: ServerKeyExchange;
        case certificate_request: CertificateRequest;
        case server_hello_done:   ServerHelloDone;
        case certificate_verify:  CertificateVerify;
        case client_key_exchange: ClientKeyExchange;
        case finished:            Finished;
        } body;
} Handshake;
```

Listing 4.3: Handshake Struct [3]

The handshake protocol messages must be sent in a specific sequence; deviating from this order causes a critical error. However, optional handshake messages may be skipped. The Certificate message is an exception, exchanged twice during the handshake (first from the server to the client, then from the client to the server), but detailed only in its initial context. Unlike other messages, the `HelloRequest` is not bound by these ordering rules and may be sent at any time. If the client receives a `HelloRequest` during a handshake, it should be ignored.

4.1.1. Hello Request

A `HelloRequest` message, shown in Listing 4.4, notifies the client to restart the negotiation process. The client should reply with a `ClientHello` message when appropriate, see Figure 4.1. This message merely initiates a new negotiation and does not determine whether the sender is the client or server. Servers should avoid sending a HelloRequest right after the client's initial connection, as it is up to the client to send a `ClientHello` at that time.

If a session negotiation is in progress, the client will disregard the `HelloRequest`. The client may also choose to ignore the message if it does not want to renegotiate or respond with a no_renegotiation alert. Handshake messages take precedence over application data, so the new negotiation should begin after receiving only a few records from the client. If the server sends a `HelloRequest` and does not get a `ClientHello` in return, it might close the connection with a fatal alert.

```
struct { } HelloRequest;
```

Listing 4.4: HelloRequest Struct [3]

4.1.2. Client Hello

Upon establishing initial connection with a server, the client must initiate communication by sending the `ClientHello` as its primary message, shown in Listing 4.5. Additionally, the client may send a `ClientHello` in response to a `HelloRequest` or independently to propose renegotiation of security parameters within an existing connection.

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

Listing 4.5: ClientHello Struct [3]

The `ClientHello` struct contains several fields that are crucial for establishing a secure connection in the Transport Layer Security (TLS) protocol:

- `client_version`: specifies the highest TLS protocol version supported by the client.
- `random`: a structure containing the current time and date and client's random value of 28 bytes.

```
struct {
    uint32 gmt_unix_time;
    opaque random_bytes[28];
} Random;
```

Listing 4.6: Random Struct [3]

- `session_id`: the identifier of a session that the client intends to use for this connection. This field is left blank if no session_id is available or if the client opts to create new security parameters.
- `cipher_suites`: specifies the list of cryptographic algorithms (cipher suites) supported by the client, in order of preference.
- `compression_methods`: specifies the compression algorithms supported by the client for data compression during the session.
- `extensions`: clients have the option to request additional functionality from servers by transmitting data in the extensions field.

4.1.3. Server Hello

Whenever the server receives a `ClientHello` message, it will respond with `ServerHello`, as shown in Listing 4.7, if it successfully identifies a compatible set of algorithms. If no such match is found, the server will instead reply with a handshake failure alert.

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ServerHello;
```

Listing 4.7: ServerHello Struct [3]

The `ServerHello` struct contains several fields that are crucial for establishing a secure connection in the Transport Layer Security (TLS) protocol:

- `server_version`: this field will hold the smaller value between what the client proposed in the client hello and the maximum that the server can support.
- `random`: this structure must be generated by the server autonomously, distinct from the ClientHello.random.
- `session_id`: serves as the session identifier for the current connection. If the client provides a non-empty `session_id`, the server checks its cache for a match to resume the session. If found, it uses the same `session_id` for resumed sessions; otherwise, it generates a new one.
- `cipher_suite`: the server chooses the single cipher suite found from the list in the ClientHello.cipher_suites.
- `compression_method`: the server selects the single compression algorithm from the list in the ClientHello.compression_methods.
- `extensions`: the list of the available extensions.

4.1.4. Certificate

When certificates are utilized for authentication in the chosen key exchange method, a `Certificate` message, shown in Listing 4.8, is required to be transmitted by the server. This message consistently follows the `ServerHello` message. The purpose of this message is to deliver the server's certificate chain to the client, ensuring the certificate aligns properly with the key exchange algorithm of the negotiated cipher suite and any agreed-upon extensions.

```
struct {
    ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
```

Listing 4.8: Certificate Struct [3]

The `certificate_list` parameter is essential for managing and validating a sequence of digital certificates within a communication protocol. It plays a critical role in ensuring the integrity and authenticity of certificates, which is vital for establishing secure connections between entities.

- `certificate_list`: comprises a series of certificates organized in a specific order. The list begins with the sender's certificate, which is followed by certificates that directly verify the one before it. This sequential arrangement is crucial for maintaining a chain of trust. Notably, the root certificate, which is self-signed, can be excluded from this sequence. This exclusion operates under the premise that the recipient already possesses the root certificate necessary for the validation process.

4.1.5. Server Key Exchange

This message, shown in Listing 4.9, is dispatched right after the server's `Certificate` message, or following the `ServerHello` message in the case of an anonymous negotiation.

The purpose of this message is to provide the client with the necessary cryptographic information for establishing the Pre-master secret. This includes a Diffie-Hellman public key for completing the key exchange or a public key for another cryptographic algorithm.

```
struct {
    select (KeyExchangeAlgorithm) {
        case dh_anon:
            ServerDHParams params;
        case dhe_dss:
        case dhe_rsa:
            ServerDHParams params;
            digitally-signed struct {
                opaque client_random[32];
                opaque server_random[32];
                ServerDHParams params;
            } signed_params;
        case rsa:
        case dh_dss:
        case dh_rsa:
            struct {} ;
            /* message is omitted for rsa, dh_dss, and dh_rsa */
        /* may be extended, e.g., for ECDH -- see [TLSECC] */
    };
} ServerKeyExchange;
```

Listing 4.9: ServerKeyExchange Struct [3]

The `ServerKeyExchange` struct incorporates multiple fields essential for securely establishing a connection in the Transport Layer Security (TLS) protocol:

- `params`: contains the server's key exchange parameters. Its content varies depending on the specific key exchange algorithm used.
- `signed_params`: in non-anonymous key exchanges, this field includes a digital signature covering the server's key exchange parameters. It includes `client_random`, `server_random`, and the `ServerDHParams` to ensure the integrity and authenticity of the exchanged parameters.

4.1.6. Certificate Request

In cases where the server is non-anonymous, it retains the option to solicit a certificate from the client, provided it aligns with the chosen cipher suite's requirements, as shown in Listing 4.10. This request, when dispatched, promptly succeeds the `ServerKeyExchange` message (should it be transmitted; otherwise, it trails the server's `Certificate` message).

```
struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    SignatureAndHashAlgorithm supported_signature_algorithms<2^16-1>;
    DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
```

Listing 4.10: CertificateRequest Struct [3]

The `CertificateRequest` structure includes several fields crucial for establishing a secure connection within the Transport Layer Security (TLS) protocol:

- `certificate_type`: a compilation of the varieties of certificates that the client could potentially present.
- `supported_signature_algorithms`: a roster of hash/signature algorithm pairs that the server can verify, arranged from most to least preferred.
- `certificate_authorities`: n list of distinguished names in DER-encoded format, denoting acceptable certificate authorities.

4.1.7. Server Hello Done

This message, shown in Listing 4.11, marks the completion of the server's initial communication sequence, signaling that the server has concluded its part of the key exchange preparation. The server will then pause, anticipating the client's next message.

The purpose of this message is to inform the client that the server's contribution to the key exchange is finished, and the client can now continue with its steps. Upon receiving this notification, the client is expected to confirm the validity of the server's certificate, if applicable, and verify that the server's hello parameters are appropriate.

```
struct { } ServerHelloDone;
```

Listing 4.11: ServerHelloDone Struct [3]

4.1.8. Certificate Verify

This message, shown in 4.12, serves to conclusively verify a client's certificate. It is dispatched exclusively subsequent to a client certificate endowed with signing capabilities (excluding those fixed Diffie-Hellman parameters). Upon dispatch, it is mandatory for this message to directly succeed the client key exchange message.

```
struct {
    digitally-signed struct {
        opaque handshake_messages[handshake_messages_length];
    }
} CertificateVerify;
```

Listing 4.12: CertificateVerify Struct [3]

The term `handshake_messages` encompasses all messages within the handshake process, transmitted or received, beginning from the `ClientHello` up to (but excluding) the current message. This includes the type and length fields of each handshake message.

4.1.9. Client Key Exchange

This message, shown in 4.13, is dispatched by the client under specific conditions. If a client certificate message is transmitted, the client key exchange message must follow immediately. In scenarios where no client certificate message is sent, the client key exchange message must be the initial message sent by the client after the reception of the ServerHelloDone message.

The purpose of this message is to establish the Pre-master secret. This is achieved through either the transmission of an RSA-encrypted secret or Diffie-Hellman parameters that enable both parties to agree on the Pre-master secret. For clients using an ephemeral Diffie-Hellman exponent, the message includes the client's Diffie-Hellman public value. In cases where the client provides a certificate with a static Diffie-Hellman exponent for fixed_dh client authentication, the message must still be sent but will remain empty.

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa:
            EncryptedPreMasterSecret;
        case dhe_dss:
        case dhe_rsa:
        case dh_dss:
        case dh_rsa:
        case dh_anon:
            ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

Listing 4.13: ClientKeyExchange Struct [3]

4.1.10. Finished

After a change cipher spec message, a `Finished` message is promptly transmitted to confirm the success of both the key exchange and authentication processes. It is crucial that a change cipher spec message is received between other handshake messages and the transmission of the `Finished` message.

The `Finished` message represents the initial communication secured using the newly agreed-upon algorithms, keys, and confidential information. Recipients of `Finished` messages are required to ensure the accuracy of their contents. Once both sides have sent and verified their respective `Finished` messages, confirming them with their peers, they are authorized to begin the exchange of application data over the connection.

```
struct {
    opaque verify_data[verify_data_length];
} Finished;
```

Listing 4.14: Finished Struct [3]

4.2 Elliptic-curve Diffie–Hellman

Elliptic Curve Diffie-Hellman (ECDH) [5] operates as a cryptographic protocol designed for secure key exchange between two parties over an insecure communication channel. It employs the principles of elliptic curve cryptography (ECC) to enable the derivation of a shared secret, which can subsequently be used for encryption, without the need to directly transmit the secret itself. The strength of ECDH lies in its use of elliptic curves, which provide robust security with smaller key sizes compared to traditional cryptographic methods such as the classical Diffie-Hellman (DH) algorithm. This attribute not only enhances security but also significantly improves computational efficiency and performance, making ECDH a preferred choice in many modern cryptographic applications.

In practical terms, ECDH involves each party generating a public-private key pair based on elliptic curve parameters. The public keys are exchanged between the parties, while the private keys remain confidential. Each party then combines their private key with the received public key to compute a shared secret. Due to the mathematical properties of elliptic curves, the resulting shared secret is identical for both parties, despite their use of different key pairs. This shared secret can then be employed to generate symmetric encryption keys, enabling secure communication.

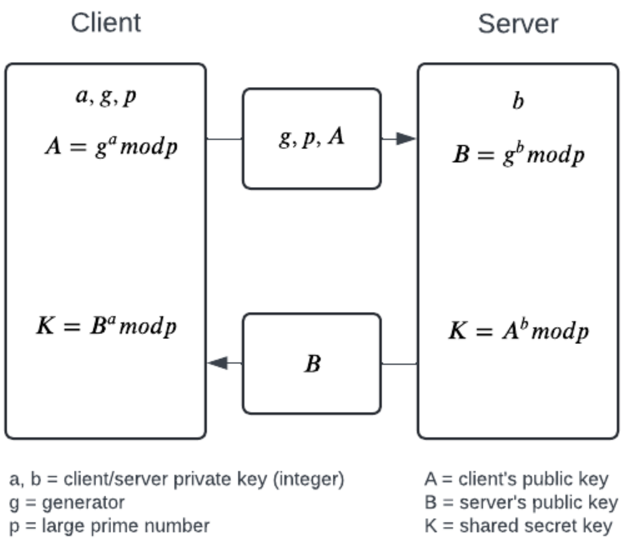


Figure 4.2: Elliptic-curve Diffie-Hellman diagram

4.2.1. Mathematical Concepts and Applications

Elliptic Curve Diffie-Hellman (ECDH) hinges on the mathematical principles of elliptic curves, which are described by specific types of cubic equations. The standard form of an elliptic curve equation used in cryptography is:

$$y^2 = x^3 + ax + b$$

where a and b are coefficients that define the curve's shape. For cryptographic purposes, it is essential that the curve is non-singular, which means that the following condition must be met to avoid any cusps or self-intersections:

$$4a^3 + 27b^2 \neq 0$$

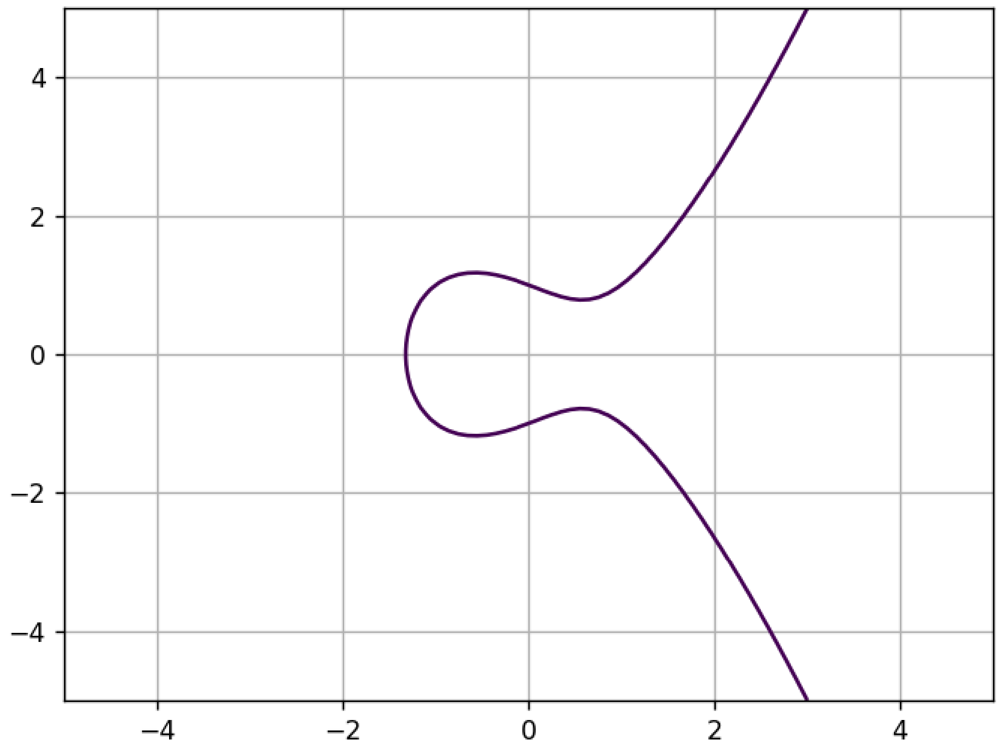


Figure 4.3: Representation of elliptic curve equation $y^2 = x^3 - x + 1$

4.2.2. Point Operations

Operations on elliptic curves include point addition and point doubling:

- **Point Addition**: to add two distinct points P and Q on the curve, draw a line connecting them. This line will intersect the curve at a third point, which is then reflected across the x-axis to yield the sum $R = P + Q$.
- **Point Doubling**: to double a point P (find $2P$), draw the tangent line to the curve at P. This line will intersect the curve at another point, which is also reflected over the x-axis to give the result $R = 2P$.

These operations are fundamental for the elliptic curve's group structure and are used in the scalar multiplication process, where a point is added to itself multiple times.

4.2.3. Discrete Logarithm Problem

The security of ECDH relies on the Elliptic Curve Discrete Logarithm Problem (ECDLP). Given a point P and a point Q such that $Q = kP$ (where k is an integer), it is computationally hard to determine k knowing only P and Q. This difficulty forms the cornerstone of elliptic curve cryptography's security.

4.2.4. Efficiency and Security

ECDH is favored in modern cryptographic applications because it provides strong security with high efficiency. The key sizes in elliptic curve cryptography are significantly smaller than those needed in traditional systems like RSA for comparable security levels. For instance, a 256-bit elliptic curve key offers equivalent security to a 3072-bit RSA key. This efficiency translates into faster computations, reduced storage requirements, and lower power consumption.

However, it's important to note that the security of ECDH, like all current cryptographic systems, relies on the assumption that certain mathematical problems, such as the discrete logarithm problem on elliptic curves, are computationally hard to solve. The advent of quantum computers poses a potential threat to this security assumption. Quantum computers have the potential to efficiently solve certain mathematical problems that underpin the security of many cryptographic protocols, including ECDH. Specifically, quantum computers could break the discrete logarithm problem on elliptic curves using algorithms like Shor's algorithm, rendering current implementations vulnerable.

As a result, there is ongoing research and development in post-quantum cryptography to develop cryptographic algorithms that are resistant to quantum attacks. These algorithms aim to provide security against both classical and quantum computers, ensuring the long-term confidentiality and integrity of digital communications in a post-quantum computing era. Thus, while ECDH remains robust against classical computational threats, the potential emergence of quantum computing underscores the need for continued advancements in cryptographic techniques to maintain secure communication channels in the future.

CHAPTER 5

Hybrid Post-Quantum Transport Layer Security protocol

Hybrid Post-Quantum Transport Layer Security (Hybrid PQ-TLS) [4] was introduced by Matt Campagna and Eric Crockett from Amazon Web Services. The protocol aims to enhance the security of key exchanges in the Transport Layer Security (TLS) protocol by incorporating both classical and post-quantum cryptographic methods.

Hybrid key exchange involves the execution of two independent key exchanges, with the shared secrets from both being combined using a Pseudo Random Function (PRF). This method ensures that the derived secret retains the security level of the stronger key exchange method. New hybrid key exchange schemes have been tailored for the TLS 1.2 protocol, combining Elliptic Curve Diffie-Hellman (ECDH) with a post-quantum key encapsulation method (PQ KEM) utilizing the existing TLS PRF.

Additions to TLS support PQ Hybrid Key Exchanges, specifically designed for compatibility with most third-round candidates from the NIST Call for Proposals. Cipher suites are defined for a limited subset of potential hybrid key agreement methods, notably combining ECDHE with BIKE, Kyber, or SIKE.

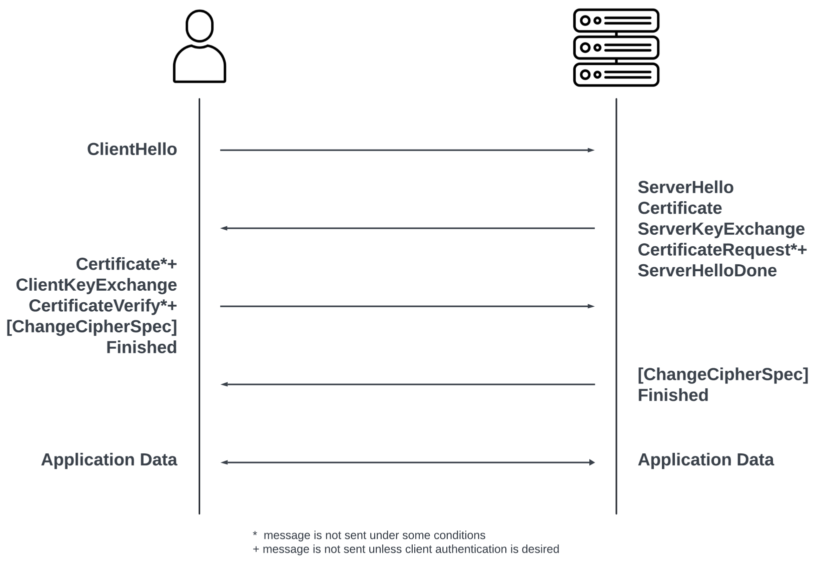


Figure 5.1: Hybrid Post-Quantum TLS protocol [4]

5.1 Hybrid Key Exchanges

The Hybrid Post-Quantum Transport Layer Security (Hybrid PQ-TLS) protocol introduces several hybrid key exchange schemes that combine Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) with various post-quantum key encapsulation methods (PQ KEM). These hybrid schemes, namely ECDHE_BIKE, ECDHE_KYBER, and ECDHE_SIKE, are tailored to fortify the security of the TLS 1.2 protocol by ensuring the derived secret matches the resilience of the strongest key exchange method employed.

Hybrid Key Exchange Scheme Name	Description
ECDHE_BIKE	ECDHE and BIKE
ECDHE_KYBER	ECDHE and Kyber
ECDHE_SIKE	ECDHE and SIKE

Table 5.1: Hybrid Key Exchange Schemes [4]

5.1.1. ECDHE-BIKE

This scheme combines ECDHE with Bit Flipping Key Encapsulation (BIKE) [6], a post-quantum cryptographic algorithm designed to resist quantum computer attacks. By integrating ECDHE with BIKE, this hybrid approach capitalizes on the security strengths of both classical and post-quantum cryptography. ECDHE ensures efficient and widely-accepted security for current systems, while BIKE offers protection against potential future quantum threats.

ECDHE is a well-established key exchange protocol that provides forward secrecy, ensuring that session keys are not compromised even if long-term private keys are compromised in the future. This makes it a robust choice for secure communications over untrusted networks. Moreover, BIKE [9] is part of the NIST Post-Quantum Cryptography Standardization project and is based on the hardness of decoding random linear codes, a problem believed to be resistant to attacks from both classical and quantum computers.

BIKE operates by encapsulating a randomly chosen symmetric key, which is then securely transmitted to the recipient. This symmetric key is subsequently used for encrypting the actual data. The Bit Flipping Key Encapsulation method ensures that any attempt to decode the key without the correct private key will result in a high probability of error, thus thwarting potential attackers.

5.1.2. ECDHE-KYBER

This scheme merges Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) with Kyber [7], a post-quantum cryptographic algorithm engineered to withstand attacks from quantum computers. By combining ECDHE with Kyber, this hybrid approach takes advantage of the security benefits of both classical and post-quantum cryptography. ECDHE ensures efficient, widely-accepted security for current systems, while Kyber provides protection against future quantum threats.

ECDHE is a time-tested key exchange protocol that offers forward secrecy, meaning that even if long-term private keys are compromised in the future, past session keys remain secure. This characteristic makes it a reliable choice for secure communications over potentially untrusted networks. In contrast, Kyber is a lattice-based key encapsulation mechanism included in the NIST Post-Quantum Cryptography Standardization project. It relies on the difficulty of the Learning With Errors (LWE) problem, which is believed to be resistant to both classical and quantum attacks.

Kyber encapsulates a randomly generated symmetric key, which is then securely transmitted to the recipient. This key is subsequently used for data encryption. The use of lattice-based cryptography in Kyber ensures high efficiency and low computational overhead, making it suitable for a broad range of applications.

5.1.3. ECDHE-SIKE

This scheme integrates Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) with Supersingular Isogeny Key Encapsulation (SIKE) [8], a post-quantum cryptographic algorithm designed to withstand quantum computer attacks. By combining ECDHE with SIKE, this hybrid approach leverages the security strengths of both classical and post-quantum cryptography. ECDHE provides efficient and widely-recognized security for contemporary systems, while SIKE offers protection against future quantum threats.

ECDHE is a well-established key exchange protocol known for providing forward secrecy. This ensures that even if long-term private keys are compromised, past session keys remain secure. This property makes ECDHE a reliable choice for secure communications over untrusted networks. In contrast, SIKE is based on the hardness of computing isogenies between supersingular elliptic curves, a problem believed to be resistant to attacks by both classical and quantum computers. SIKE is one of the candidates in the NIST Post-Quantum Cryptography Standardization project.

SIKE encapsulates a randomly chosen symmetric key, which is then securely transmitted to the recipient. This key is used for encrypting the actual data. The use of isogeny-based cryptography in SIKE ensures that it has relatively low computational overhead and small key sizes, making it an efficient choice for a variety of applications.

5.2 Hybrid Pre-master Secret

To elaborate further on the mechanism of combining the Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) and Post-Quantum Key Encapsulation Mechanism (KEM) secrets into a TLS 1.2 Pre-master secret [3], it's important to understand the dual nature of the key exchange process. In this hybrid approach, both the server and client play essential roles in computing shared secrets that contribute to the overall security of the session.

Firstly, the ECDHE shared secret Z, defined in RFC 8422 [11], is computed using elliptic curve cryptography. This involves each party generating their respective ephemeral keys and then performing a series of mathematical operations to derive Z, according to Figure 4.2. The security of ECDHE relies on the hardness of the elliptic curve discrete logarithm problem, ensuring that even if the communication is intercepted, an adversary cannot feasibly determine Z without solving this computationally challenging problem.

Additionally, another shared secret K is derived from underlying Post-Quantum Key Encapsulation Method. This process involves the client and server each generating key pairs and exchanging public keys to encapsulate and decapsulate the shared secret K. By combining both ECDHE shared secret Z and the PQ KEM shared secret K, the hybrid key exchange approach enhances the overall security, providing protection against both classical and quantum adversaries.

Finally, these two secrets Z and K are combined to form the Pre-master secret, ensuring that the session keys derived from this Pre-master secret inherit the security properties of both ECHDE and PQ KEM. This dual-layered approach is a critical advancement in cryptographic protocols, aiming to future-proof secure communications in the advent of quantum computing.

5.3 Hybrid PQ-TLS Handshake protocol

In Hybrid Post-Quantum TLS, several differences arise in the data structures used during the handshake process. These modifications are crucial to integrating quantum-resistant algorithms alongside traditional cryptographic methods. Understanding these structural variations is essential for implementing and maintaining the enhanced security offered by Hybrid Post-Quantum TLS. Specifically, in this context, only the modified data structures that accommodate post-quantum KEM parameters and extensions will be presented.

5.3.1. Client Hello

In Hybrid Post-Quantum TLS, the handshake process incorporates additional extensions to facilitate the use of post-quantum cryptographic algorithms. One such extension is the Supported PQ KEM Parameters extension, which can be included in the `ClientHello` (Figure 4.5) message as specified in RFC 5246 [3].

```
enum {
    pq_kem_parameters(0xFE01)
} ExtensionType;
```

Listing 5.1: ExtensionType Enumeration

The `pq_kem_parameters` extension denotes the client's supported set of post-quantum KEM parameters. It plays a crucial role in Hybrid Post-Quantum TLS by specifying which cryptographic algorithms the client can utilize.The `extension_data` field remains opaque and contains the `PQKEMParametersExtension`, which encapsulates detailed information about the specific parameters supported.

5.3.2. PQ KEM Parameters Extension

The Supported PQ KEM Parameters extension defines the structure for incorporating post-quantum cryptographic algorithms into the handshake process. According to RFC 5246 [3], the `extension_data` is specified as the following PQKEMParametersExtension:

```
enum {
    SIKE-P434-R3 (19),
    SIKE-P503-R3 (20),
    SIKE-P610-R3 (21),
    SIKE-P751-R3 (22),
    BIKE-L1-R3(25),
    BIKE-L3-R3(26),
    BIKE-L5-R3(27),
    KYBER-512-R3 (28),
    KYBER-512-90s-R3 (29)
} NamedPQKEM (2^16-1);
```

Listing 5.2: NamedPQKEM Enumeration

The identifiers `BIKE-L1-R3`, etc., indicate support for the corresponding BIKE parameters as defined in BIKE [6]. Similarly, identifiers like `SIKE1-P434-R3`, etc., indicate support for corresponding SIKE parameters as defined in SIKE [8]. Identifiers such as

KYBER-512-R3, etc., denote support for corresponding KYBER parameters as defined in Kyber [7].

```
struct {
    NamedPQKEM pq_kem_parameters_list <1..2^16-1>
} PQKEMParametersExtension;
```

Listing 5.3: PQKEMParametersExtension Struct

Within PQKEMParametersExtension, pq_kem_parameters_list arranges items based on the client's preferences, with the most favored option listed first.

5.3.3. Server Key Exchange

The Server Key Exchange message is transmitted when using hybrid key exchange algorithms that combine ECDHE with post-quantum Key Encapsulation Mechanisms (KEM). This message's purpose is to convey the server's ephemeral ECDH and post-quantum KEM public keys to the client.

```
struct {
    opaque public_key <1,...,2^24 - 1>;
} PQKEMPublicKey;
```

Listing 5.4: PQKEMPublicKey Struct

The public_key field is a byte string representation of the post-quantum KEM public key, formatted according to the specifications of the KEM implementation in use. Only uncompressed formats of these public keys are supported.

```
struct {
    NamedPQKEM named_params;
    PQKEMPublicKey public;
} ServerPQKEMParams;
```

Listing 5.5: ServerPQKEMParams Struct

The ServerKeyExchange message is augmented to include additional parameters and structures as detailed below:

```
struct {
    ServerECDHParams ecdh_params;
    ServerPQKEMParams pq_kem_params;
    Signature signed_params;
} ServerKeyExchange;
```

Listing 5.6: ServerKeyExchange Struct

In this enhanced ServerKeyExchange message, the ecdh_params field specifies the ECDHE public key along with its associated domain parameters. The pq_kem_params field designates the post-quantum KEM public key and the parameters linked to it. Additionally, the signed_params field encompasses a digital signature covering the server's key exchange parameters. The signing process utilizes the private key corresponding to the certified public key presented in the server's Certificate message.

```
digitally-signed struct {
    opaque client_random[32];
    opaque server_random[32];
    ServerDHParams ecdh_params;
    ServerPQKEMParams pq_kem_params;
} Signature;
```

Listing 5.7: Signature Struct

5.3.4. Client Key Exchange

The Client Key Exchange message is transmitted in all key exchange algorithms. Within the context of the key exchanges described in this document, it contains the client's ephemeral ECDH public key and the post-quantum KEM ciphertext value.

This message serves the purpose of conveying ephemeral data related to the client's part of the key exchange, including the post-quantum KEM ciphertext value and its ephemeral ECDH public key .

The structure of the TLS ClientKeyExchange message has been modified as outlined below:

```
struct {
    opaque ciphertext <1,..., 2^24 - 1>;
} PQKEMCiphertext;
```

Listing 5.8: PQKEMCiphertext Struct

The `ciphertext` is presented as a byte string that represents the post-quantum KEM `ciphertext`. Given that the KEM API handles the `ciphertext` byte string directly according to its internal conventions, it suffices to transmit this as a single byte string array within the protocol. This specification exclusively supports uncompressed formats for post-quantum public keys.

```
struct {
    ClientECDiffieHellmanPublic ecdh_public;
    PQKEMCiphertext ciphertext;
} ClientKeyExchange;
```

Listing 5.9: ClientKeyExchange Struct

In this modified `ClientKeyExchange` message the `ecdh_public` parameter contains the client's ephemeral ECDH public key, while the `ciphertext` parameters holds the post-quantum KEM `ciphertext`.

CHAPTER 6

Protocol Modeling and Results

In this chapter, the detailed modeling and analysis of the Hybrid Post-Quantum TLS protocol [4] are presented. Previously, the conceptual framework and theoretical underpinnings of the protocol were introduced. A reference to an earlier version of the protocol was used as a foundation for this thesis. The current study aims to improve the performance of the Hybrid Post-Quantum TLS protocol and verify additional properties that enhance its applicability and robustness.

The focus now shifts to the practical implementation and performance evaluation of the improved protocol. The objective of this chapter is to demonstrate the functioning of the Hybrid Post-Quantum TLS protocol in a simulated environment, evaluate its effectiveness, and analyze the results to draw meaningful conclusions about its viability and potential impact.

The chapter begins with an outline of the methodology employed for modeling the Hybrid Post-Quantum TLS protocol, encompassing the Syntax, Algebraic, and Behavioral modules. Each module's tools, techniques, and parameters are detailed to provide a comprehensive understanding of the modeling approach. Following this, the chapter presents the results of experimental simulations. Key performance metrics such as security robustness, computational efficiency, and overall protocol performance are analyzed in depth. These results serve to both validate the earlier theoretical assertions and offer empirical evidence supporting the enhancements introduced in the Hybrid Post-Quantum TLS protocol.

Additionally, the implications of the findings are discussed, comparisons with the previous version of the protocol [10], and potential areas for further research and improvement are suggested. By the end of this chapter, a thorough understanding of the operation of the improved Hybrid Post-Quantum TLS protocol in practice and its effectiveness in addressing the challenges posed by quantum computing advancements will be provided.

6.1 Syntax Module Specification

Various sorts are defined to categorize and represent data structures and different entities essential for its functionality. Each sort serves a distinct role in the protocol's operations, contributing to its overall security and performance.

```
--- Sorts declaration
sorts Name
      Random
```

```
          Certificate
          Session
          Scalar
          Point
          Elliptic-Curve-Key
          Post-Quantum-Public-Key
          Post-Quantum-Secret-Key
          Post-Quantum-Shared-Key
          Cipher
          Pre-Master-Secret
          Master-Secret .
```

These sorts collectively define the foundational components and data structures of the Hybrid Post-Quantum TLS protocol, facilitating its secure and efficient operation across various communication scenarios, where:

- `Name`: identifier to represent entities or persons.
- `Random`: representation of arbitrary numbers.
- `Certificate`: digital object that allows to verify identities.
- `Session`: re-authentication process.
- `Scalar`: integer used in elliptic curve exponentiations.
- `Point`: specific location of the elliptic curve function.
- `Elliptic-Curve-Key`: key-based technique for encrypting data.
- `Post-Quantum-Public-Key`: public key belonging to Post-Quantum side.
- `Post-Quantum-Secret-Key`: secret key belonging to Post-Quantum side.
- `Post-Quantum-Shared-Key`: shared key belonging to Post-Quantum side.
- `Cipher`: algorithm for performing encryption or decryption.
- `Pre-Master-Secret`: key exchanged during the handshake.
- `Master-Secret`: symmetric encryption key derived from pre-master-secret.

Once the protocol's sorts are defined, it is necessary that subsorts be specified to ensure comprehensive classification and compatibility within the system:

```
    --- Subsorts declaration
    subsort Random
            Session
            Certificate
            Scalar
            Elliptic-Curve-Key
            Post-Quantum-Public-Key
            Post-Quantum-Secret-Key
            Post-Quantum-Shared-Key
            Cipher
            Pre-Master-Secret
```

```
        Master-Secret < Msg .
subsort Name < Public .
subsort Point < Elliptic-Curve-Key .
```

`Name` is a subsort of `Public`, optimized for efficient symbolic search, while `Point` is a subsort of `Elliptic-Curve-Key`, specifically used within operations involving elliptic curve keys. These subsort declarations structure the sorts, defining hierarchical relationships crucial for organizing and processing data types within the specified protocol or system.

Every other sort is specified as a subsort of Msg, which is essential for message exchange in the protocol.

After having successfully declared the sorts and subsorts, the operators must be defined. The operators have been classified in different subgroups. The first subgroup contains the Elliptic-curve Diffie-Hellman operators:

```
--- Elliptic-curve Diffie-Hellman operators declaration
--- Scalar multiplication
op _*_ : Scalar
         Scalar ->
         Scalar [frozen assoc comm] .
--- Point generator
op p : ->
       Point .
--- Exponentiation
op exponentiation : Point
                    Scalar ->
                    Point [frozen] .
--- Server, Client and Intruder
ops server client intruder : ->
                             Name .
--- Random number generator
op random : Name
            Fresh ->
            Random [frozen] .
--- Scalar generator
op scalar : Name
            Fresh ->
            Scalar [frozen] .
--- Session generator
op session : Name
             Fresh ->
             Session [frozen] .
```

These operators define the fundamental Elliptic-curve Diffie-Hellman actions that we will use in the protocol behavior:

- `_*_`: this operator is used to represent the multiplication of two scalars, returning another scalar.
- `p`: represents a specific point from the elliptic-curve function.
- `exponentiation`: given a point and a scalar value, returning the result of doing the point to the power of the scalar.

- `server client intruder`: operators used to represent participants.
- `random`: uses a name of the person or entity calculating the value and a special number to return a random number.
- `scalar`: as well as `random` operator, takes the person or entity calculating the value and a special number to generate a scalar value.
- `session`: following the lead of `random` and `scalar` operators, takes the person or entity calculating the value and a special number to generate the session.

The following subgroup of operators are part of the Post-Quantum Key Encapsulation Mechanism:

```
--- Post-Quantum Key Encapsulation Mechanism
--- Secret key generator
op post-quantum-secret-key : Name
                             Fresh ->
                             Post-Quantum-Secret-Key [frozen] .
--- Public key generator
op post-quantum-public-key : Post-Quantum-Secret-Key ->
                             Post-Quantum-Public-Key [frozen] .
--- Shared key generator
op post-quantum-shared-key : Post-Quantum-Secret-Key
                             Post-Quantum-Secret-Key ->
                             Post-Quantum-Shared-Key [frozen] .
--- Encapsulation procedure, returning ciphertext
op encapsulation-cipher : Post-Quantum-Public-Key
                          Post-Quantum-Secret-Key ->
                          Cipher [frozen] .
--- Encapsulation procedure, returning shared key
op encapsulation-key : Post-Quantum-Public-Key
                       Post-Quantum-Secret-Key ->
                       Post-Quantum-Shared-Key [frozen] .
--- Decapsulation procedure
op decapsulation : Cipher Post-Quantum-Secret-Key ->
                   Post-Quantum-Shared-Key [frozen] .
```

These operators define the basic Post-Quantum Key Encapsulation Mechanism actions that will be used in the protocol behavior:

- `post-quantum-secret-key`: accepts a person or entity name and a specific number, then returns the secret key of the Post-Quantum side.
- `post-quantum-public-key`: returns the public key of the Post-Quantum side when given the Post-Quantum secret key.
- `post-quantum-shared-key`: generates the shared key of the Post-Quantum side from two Post-Quantum secret keys.
- `encapsulation-cipher`: produces ciphered text given the Post-Quantum public key and secret key.
- `encapsulation-key`: gives the Post-Quantum shared key using the Post-Quantum public key and secret key.

- decapsulation: retrieves the Post-Quantum shared key from ciphered text and a Post-Quantum secret key.

The next subgroup of operators makes up the general operators that we will use in the protocol:

```
--- General
--- Pre-master Secret generator
op pre-master-secret : Elliptic-Curve-Key
                       Post-Quantum-Shared-Key ->
                       Pre-Master-Secret [frozen] .
--- Master Secret generator
op master-secret : Pre-Master-Secret
                   Random Random
                   Point Cipher ->
                   Master-Secret [frozen] .
--- Certificate generator
op certificate : Name ->
                 Certificate [frozen] .
--- Signature generator
op signature : Name
               Point
               Post-Quantum-Public-Key
               Random
               Random ->
               Msg [frozen] .
--- Encryption procedure
op encryption : Master-Secret
                Msg ->
                Msg [frozen] .
--- Client encryption procedure
op client-encryption : Master-Secret
                       Msg
                       Msg
                       Msg
                       Msg
                       Msg ->
                       Msg [frozen] .
--- Server encryption procedure
op server-encryption : Master-Secret
                       Msg
                       Msg
                       Msg
                       Msg
                       Msg
                       Msg ->
                       Msg [frozen] .
--- Decryption procedure
op decryption : Master-Secret
                Msg ->
                Msg [frozen] .
--- Message concatenation
```

```
op _;_ : Msg
         Msg ->
         Msg [gather (e E) frozen] .
```

Each of these operators fulfill an essential part in order to model the protocol or intruder specifications:

- `pre-master-secret`: given an Elliptic-curve key and the Post-Quantum shared key, it generates the Pre-master Secret.
- `master-secret`: uses the Pre-master Secret, two random values from the client and the server respectively, a point from the curve and the ciphered text, to generate the Master Secret.
- `certificate`: takes the name of the person or entity to generate the certificate.
- `signature`: accepts the name of the person or entity, a point, the Post-Quantum public key and two random values from the client and the server respectively, to generate the signed message.
- `encryption`: generates the encrypted message making use of the Master Secret and the original message.
- `client-encryption`: obtains the client encrypted message using the Master Secret, and the previous messages sent in the protocol by the client.
- `server-encryption`: returns the server encrypted message using the Master Secret, and the previous messages sent in the protocol by the server.
- `decryption`: takes the Master Secret and the encrypted message to obtain the original message.
- `_;_`: used to concatenate a couple of messages into one.

The last subgroup of operators defines the Hybrid Post-Quantum TLS protocol phases to allow the communication between the server and the client:

```
--- Protocol phases
--- ClientHello
op client-hello : Random ->
                  Msg .
--- ServerHello
op server-hello : Random
                  Session ->
                  Msg .
--- ServerCertificate
op server-certificate : Certificate ->
                        Msg .
--- ServerKeyExchange
op server-key-exchange : Point
                         Post-Quantum-Public-Key
                         Msg ->
                         Msg .
--- ClientKeyExchange
```

```
op client-key-exchange : Point
                         Cipher ->
                         Msg .
--- ClientFinished
op client-finished : Msg ->
                     Msg .
--- ServerFinished
op server-finished : Msg ->
                     Msg .
```

This subgroup has the most important operators to make the protocol able to communicate successfully:

- `client-hello`: takes a random generated value to generate a message.
- `server-hello`: given a random generated value and a session, it generates a message.
- `server-certificate`: uses a certificate to generate a message.
- `server-key-exchange`: combines a point, a Post-Quantum public key, and a message to form a message.
- `client-key-exchange`: uses a point and a cipher to produce a message.
- `client-finished`: takes a message and generates another message.
- `server-finished`: uses a message to generate another message.

In the definition of all the previous operators, the attribute `frozen` is used to not attempt applying rewrites at the arguments. The attribute `assoc` refers to associativity property, involving symbols with any combination. The attribute `comm` represents commutativity. Finally the attribute `gather` means association to the right.

6.2 Algebraic Module Specification

The following equations have been defined to represent the equivalences and properties within cryptographic protocols, specifically targeting elliptic-curve Diffie–Hellman (ECDH) and Post-Quantum key encapsulation mechanisms. These algebraic properties ensure the correctness and consistency of cryptographic operations within the specified protocol.

```
--- Elliptic-curve Diffie-Hellman algebraic property
--- Exponentiation equation
eq exponentiation(
      exponentiation(
         PointC:Point,
         Scalar1:Scalar),
       Scalar2:Scalar) =
   exponentiation(
      PointC:Point,
      Scalar1:Scalar * Scalar2:Scalar) [variant] .
```

This equation captures the associativity of exponentiation in the context of Elliptic-curve operations. It states that exponentiating a point by two consecutive scalars can be simplified to exponentiating the point by the product of the two scalars

```
--- Post-Quantum key encapsulation mechanism algebraic properties
--- Encapsulation key equation
eq encapsulation-key(
      post-quantum-public-key(
         secret-key1:Post-Quantum-Secret-Key),
      secret-key2:Post-Quantum-Secret-Key) =
   post-quantum-shared-key(
      secret-key1:Post-Quantum-Secret-Key,
      secret-key2:Post-Quantum-Secret-Key) [variant] .
--- Decapsulation equation
eq decapsulation(
      encapsulation-cipher(
         post-quantum-public-key(
            secret-key1:Post-Quantum-Secret-Key),
            secret-key2:Post-Quantum-Secret-Key),
            secret-key1:Post-Quantum-Secret-Key) =
   post-quantum-shared-key(
      secret-key1:Post-Quantum-Secret-Key,
      secret-key2:Post-Quantum-Secret-Key) [variant] .
```

The last two equations specify the algebraic properties for post-quantum key encapsulation mechanisms, ensuring consistency in cryptographic operations. The encapsulation key equation states that combining a public key derived from one post-quantum secret key with another secret key results in a shared key. The decapsulation equation indicates that decapsulating a cipher, formed by encapsulating a public key (derived from one secret key) with another secret key, using the original secret key will yield the same shared key. These properties ensure the correctness of the encapsulation and decapsulation processes.

```
--- General algebraic property
eq decryption(
      master-secret1:Master-Secret,
      encryption(
         master-secret1:Master-Secret,
         msg1:Msg)) =
   msg1:Msg [variant] .

eq decryption(
      master-secret1:Master-Secret,
      client-encryption(
         master-secret1:Master-Secret,
         msg1:Msg,
         msg2:Msg,
         msg3:Msg,
         msg4:Msg,
         msg5:Msg)) =
   msg1:Msg [variant] .
```

```
eq decryption(
      master-secret1:Master-Secret,
      server-encryption(
         master-secret1:Master-Secret,
         msg1:Msg,
         msg2:Msg,
         msg3:Msg,
         msg4:Msg,
         msg5:Msg,
         msg6:Msg)) =
   msg1:Msg [variant] .
```

The provided equations define general algebraic properties of decryption within cryptographic protocols. The first equation ensures that decrypting a message encrypted with a master secret key retrieves the original message. The second equation specifies that decrypting a client-encrypted message, which includes the master secret key and multiple messages, will yield the first message. The third equation states that decrypting a server-encrypted message, using the master secret key along with multiple messages, also results in retrieving the first message.

6.3 Behavior Module Specification

This section outlines the behavior module specification, focusing on the detailed definition of the protocol as implemented on both the client and server's sides. It establishes the sequence of interactions, message exchanges, and state transitions that occur during the execution of the cryptographic protocols, particularly emphasizing the elliptic-curve Diffie–Hellman (ECDH) and Post-Quantum key encapsulation mechanisms. These behavioral specifications ensure that the protocol operations adhere to the designed cryptographic processes, maintaining security and integrity across the client-server communications.

Additionally, this section explains the intruder capabilities and defines the potential attacks on the protocol. By understanding the range of actions an intruder can perform and the specific vulnerabilities they may exploit, we can better design robust countermeasures to protect against such threats. This comprehensive approach ensures a thorough evaluation and enhancement of protocol security.

The first phase to be modeled in this specification is the `ClientHello` message. This initial message is crucial as it sets the parameters for the subsequent communication and initiates the protocol handshake. Fresh variables (uniquely generated) are explicitly identified in this section.

```
--- Client side
--- ClientHello
+ (client-hello(
    random(
        ClientC,
        random1:Fresh
    )
  )),
--- Server side
--- ClientHello
- (client-hello(
    NonceC
  )),
```

From the given structure of `ClientHello` (Figure 4.5) and the extension of the Post-Quantum side (Figure 5.1) it has been decided to only include the `random` parameter, discarding the other parameters. The client's side generates the random generated value, which is received as `NonceC` by the server, defining that it is the Nonce generated by the client. The Nonce is generated by defining the entity that is going to generate the value and the Fresh variable `random1`.

```
--- Client side
--- ServerHello
- (server-hello(
    NonceS,
    SessionS
  )),

--- Server side
--- ServerHello
+ (server-hello(
    random(
       ServerS,
       random1':Fresh
    ),
    session(
        ServerS,
        random2':Fresh
    )
  )),
```

Given the structure of `ServerHello`, shown in Figure 4.7, only the `random` and `session_id` parameters have been included in the model. The client receives the `NonceS` variable that represents the Nonce generated by the server using the entity name and a Fresh variable. The `SessionS` makes reference to the session generated by the server, which is created using the entity's server name and another Fresh variable.

```
--- Client side
--- ServerCertificate
- (server-certificate(
    certificate(
        ServerC
    )
  )),

--- Server side
--- ServerCertificate
+ (server-certificate(
    certificate(
        ServerS
    )
  )),
```

Relying in the structure of `ServerCertificate`, shown in Figure 4.8, the parameter `certificate_list` has been modeled as `Certificate`, which is generated by the server using the entity name `ServerS` and is received by the client generating the `Certificate` with the entity name `ServerC`.

```
--- Client side
--- ServerKeyExchange
- (server-key-exchange(
    PointS,
    Post-Quantum-Public-KeyS,
    signature(
        ServerC,
        PointS,
        Post-Quantum-Public-KeyS,
        random(
            ClientC,
            random1:Fresh
        ),
        NonceS
    )
  )),

--- Server side
--- ServerKeyExchange
+ (server-key-exchange(
    exponentiation(
        p,
        scalar(
            ServerS,
            random3':Fresh
        )
    ),
    post-quantum-public-key(
        post-quantum-secret-key(
            ServerS,
            random4':Fresh
        )
    ),
    signature(
        ServerS,
        exponentiation(
            p,
            scalar(
                ServerS,
                random3':Fresh
            )
        ),
        post-quantum-public-key(
            post-quantum-secret-key(
                ServerS,
                random4':Fresh
            )
        ),
        NonceC,
        random(
            ServerS,
            random1':Fresh
        )
    )
  )),
```

The ServerKeyExchange message, shown in Listing 4.9, has been modified by the Post-Quantum extension, shown in Listing 5.6. The parameter ecdh_params has been represented as PointS in the client's side, which is generated in the server's side by doing the exponentiation of a point and a scalar value, where the scalar value is generated given the entity name, in this case ServerS and a fresh value. The following parameter pq_kem_params is received as Post-Quantum-Public-KeyS in the client's side which is generated by the server by generating the Post-Quantum secret key with the name of the entity, which is the ServerS and a fresh value, where we obtain the Post-Quantum public key. The last parameter refers to the signed_params, which includes both of the parameters that have already been described, adding the both the NonceC which was generated by the client in the ClientHello message and the NonceS created in the ServerHello message.

```
--- Client side
--- ClientKeyExchange
+ (client-key-exchange(
    exponentiation(
        p,
        scalar(
            ClientC,
            random2:Fresh
        )
    ),
    encapsulation-cipher(
        Post-Quantum-Public-KeyS,
        post-quantum-secret-key(
            ClientC,
            random3:Fresh
        )
    )
  )),

--- Server side
--- ClientKeyExchange
- (client-key-exchange(
    PointC,
    CipherC
  )),
```

Given the structure of ClientKeyExchange, shown in Listing 4.13, using the modifications described in the Post-Quantum version, shown in Listing 5.9, both parameters have been included. The parameter `ecdh_public` has been generated by the client making use of the `exponentiation` operator, which takes a point and a scalar value as a parameter, being received as `PointC` by the server, representing a point generated by the client. The last parameter `ciphertext`, that is represented making use of the `encapsulation-cipher` operator, which includes the Post-Quantum public key and the Post-Quantum secret key. This is received as `CipherC` by the server.

```
--- Client side
--- ClientFinished
+ (client-finished(
    client-encryption(
        master-secret(
            pre-master-secret(
                exponentiation(
                    PointS,
                    scalar(
                        ClientC,
                        random2:Fresh
                    )
                ),
                encapsulation-key(
                    Post-Quantum-Public-KeyS,
                    post-quantum-secret-key(
                        ClientC,
                        random3:Fresh
                    )
                )
            ),
            random(
                ClientC,
                random1:Fresh
            ),
            NonceS,
            exponentiation(
                p,
                scalar(
                    ClientC,
                    random2:Fresh
                )
            ),
            encapsulation-cipher(
                Post-Quantum-Public-KeyS,
                post-quantum-secret-key(
                    ClientC,
                    random3:Fresh
                )
            )
        ),
        --- all the previous messages will be included also
        --- [client-hello, server-hello, server-certificate,
        ---  server-key-exchange, client-key-exchange]
  )),
--- Server side
--- ClientFinished
- (client-finished(
    client-encryption(
        master-secret(
            pre-master-secret(
                exponentiation(
                    PointC,
                    scalar(
                        ServerS,
                        random3':Fresh
                    )
                ),
                decapsulation(
                    CipherC,
                    post-quantum-secret-key(
                        ServerS,
                        random4':Fresh
                    )
                )
            ),
            NonceC,
            random(
                ServerS,
                random1':Fresh
            ),
            PointC,
            CipherC
        ),
        --- all the previous messages will be included also
        --- [client-hello, server-hello, server-certificate,
        ---  server-key-exchange, client-key-exchange]
    )
  )),
```

As described in the section Hybrid Pre-master secret 5.2, the Pre-master secret will be formed by ECDHE shared secret Z, and PQ KEM shared secret K. The shared secret Z is represented by the client as the exponentiation of `PointS`, which is the point that was previously received by the server, and a scalar value generated by the client. The shared secret K is obtained as the result of making use of the `encapsulation-key` operator using the Post-Quantum public key and the Post-Quantum secret key. Once the Pre-master secret has been generated, the Master secret will be also formed by the nonces and the values that generated both ECDHE and PQ KEM shared secrets, corresponding to the Nonce generated in the `client-hello`, the Nonce received in the `server-hello`, and the shared keys generated in the `client-key-exchange` message. Once the Master secret is formed, in order to include the `verify_data` parameter from the `Finished` struct, shown in Listing 4.14, all previous messages sent and received will be included. The server will receive this message, where inside the Pre-master secret it will receive the Post-Quantum side as a encapsulated message, so the `decapsulation` operator is used to decapsulate the content.

```
--- Client side
--- ServerFinished
- (server-finished(
    server-encryption(
        master-secret(
            pre-master-secret(
                exponentiation(
                    PointS,
                    scalar(
                        ClientC,
                        random2:Fresh
                    )
                ),
                encapsulation-key(
                    Post-Quantum-Public-KeyS,
                    post-quantum-secret-key(
                        ClientC,
                        random3:Fresh
                    )
                )
            ),
            random(
                ClientC,
                random1:Fresh
            ),
            NonceS,
            exponentiation(
                p,
                scalar(
                    ClientC,
                    random2:Fresh
                )
            ),
            encapsulation-cipher(
                Post-Quantum-Public-KeyS,
                post-quantum-secret-key(
                    ClientC,
                    random3:Fresh
                )
            )
        )
        --- all the previous messages will be included also
        --- [client-hello, server-hello, server-certificate,
        ---  server-key-exchange, client-key-exchange,
        ---  client-finished]
    )
  ))
```

```
--- Server side
--- ServerFinished
+ (server-finished(
    server-encryption(
        master-secret(
            pre-master-secret(
                exponentiation(
                    PointC,
                    scalar(
                        ServerS,
                        random3':Fresh
                    )
                ),
                decapsulation(
                    CipherC,
                    post-quantum-secret-key(
                        ServerS,
                        random4':Fresh
                    )
                )
            ),
            NonceC,
            random(
                ServerS,
                random1':Fresh
            ),
            PointC,
            CipherC
        )
        --- all the previous messages will be included also
        --- [client-hello, server-hello, server-certificate,
        ---  server-key-exchange, client-key-exchange,
        ---  client-finished]
    )
  ))
```

In order to complete the `ServerFinished` struct, shown in Listing 4.14, all the data will be verified again, which will result in copying the `ClientFinished` message, in addition to including the an extra copy of the `ClientFinished` message, to verify all the messages that have been exchanged during the protocol process.

6.4 Intruder Capabilities

The intruder capabilities are defined in the `DOLEV-YAO` equation, which is also specified in the behavior module. This equation provides what actions can do the intruder when performing attacks to the protocol. There are multiple actions that can be performed by the intruder in the Hybrid Post-Quantum TLS protocol.

```
:: nil :: [nil | -(Msg1 ; Msg2),
                 +(Msg1),
           nil] &
```

```
:: nil :: [nil | -(Msg1 ; Msg2),
                 +(Msg2),
           nil] &
:: nil :: [nil | -(Msg1), -(Msg2),
                 +(Msg1 ; Msg2),
           nil] &
```

The first part of the DOLEV-YAO equations means that when the intruder receives the concatenation of one message with another, independently of the order of the message, the intruder can retrieve the information of any of the messages, and when the intruder gets both messages separately, he can merge them to get the concatenation.

```
:: nil :: [nil | -(client-hello(Random1)),
                 +(Random1),
           nil] &
:: nil :: [nil | -(Random1),
                 +(client-hello(Random1)),
           nil] &
```

When the intruder receives the client-hello message, he will be able to retrieve the Random1 value generated during the exchange, as well as if he intercepts the Random1 value, he can retrieve the entire client-hello message.

```
:: nil :: [nil | -(server-hello(Random1,
                               SessionC)),
                 +(Random1),
           nil] &
:: nil :: [nil | -(server-hello(Random1,
                               SessionC)),
                 +(SessionC),
           nil] &
:: nil :: [nil | -(Random1),
                 -(SessionC),
                 +(server-hello(Random1,
                               SessionC)),
           nil] &
```

When the intruder intercepts the server-hello message, which includes the Random1 and SessionC values, he can separately extract both Random1 and SessionC. Additionally, if the intruder intercepts both the Random1 and SessionC values separately, he can reconstruct the entire server-hello message.

```
:: nil :: [nil | -(server-certificate(Certificate1)),
                 +(Certificate1),
           nil] &
:: nil :: [nil | -(Certificate1),
                 +(server-certificate(Certificate1)),
           nil] &
```

When the intruder intercepts the server-certificate message, which contains the Certificate1, he is able to retrieve the Certificate1 value. Conversely, if the intruder already has the Certificate1 value, he can recreate the complete server-certificate message.

```
:: nil :: [nil | -(server-key-exchange(PointC,
                                       Post-Quantum-Public-KeyC,
                                       Msg1)),
                 +(PointC),
           nil] &
:: nil :: [nil | -(server-key-exchange(PointC,
                                       Post-Quantum-Public-KeyC,
                                       Msg1)),
                 +(Post-Quantum-Public-KeyC),
           nil] &
:: nil :: [nil | -(server-key-exchange(PointC,
                                       Post-Quantum-Public-KeyC,
                                       Msg1)),
                 +(Msg1),
           nil] &
:: nil :: [nil | -(PointC),
                 -(Post-Quantum-Public-KeyC),
                 -(Msg1),
                 +(server-key-exchange(PointC,
                                       Post-Quantum-Public-KeyC,
                                       Msg1)),
           nil] &
```

When the intruder intercepts the `server-key-exchange` message, including `PointC`, `Post-Quantum-Public-KeyC`, and `Msg1`, he can extract each of these components individually. Conversely, if the intruder already has `PointC`, `Post-Quantum-Public-KeyC`, and `Msg1`, he can reconstruct the entire `server-key-exchange` message. When the intruder intercepts the `server-key-exchange` message, he can extract each of these components individually. Conversely, if the intruder already has `PointC`, `Post-Quantum-Public-KeyC`, and `Msg1`, he can reconstruct the entire `server-key-exchange` message.

```
:: nil :: [nil | -(client-key-exchange(PointC,
                                       CipherS)),
                 +(PointC),
           nil] &
:: nil :: [nil | -(client-key-exchange(PointC,
                                       CipherS)),
                 +(CipherS),
           nil] &
:: nil :: [nil | -(PointC),
                 -(CipherS),
                 +(client-key-exchange(PointC,
                                       CipherS)),
           nil] &
```

When the intruder intercepts the `client-key-exchange` message containing `PointC` and `CipherS`, they can extract each component individually as shown by the first two processes. Conversely, if the intruder already has `PointC` and `CipherS`, they can reconstruct the entire `client-key-exchange` message as shown by the third process.

```
:: nil :: [nil | -(client-finished(Msg1)),
                 +(Msg1),
              nil] &
    :: nil :: [nil | -(Msg1),
                     +(client-finished(Msg1)),
              nil] &
```

When the intruder intercepts the `client-finished` message containing `Msg1`, they can extract `Msg1` as shown by the first process. Conversely, if the intruder already has `Msg1`, they can reconstruct the entire `client-finished` message using `Msg1` as shown by the second process.

```
    :: nil :: [nil | -(server-finished(Msg1)),
                     +(Msg1),
              nil] &
    :: nil :: [nil | -(Msg1),
                     +(server-finished(Msg1)),
              nil] &
```

When the intruder intercepts the `server-finished` message containing `Msg1`, they can extract `Msg1` as shown by the first process. Conversely, if the intruder already has `Msg1`, they can reconstruct the entire `server-finished` message using `Msg1` as shown by the second process.

```
    :: random1 :: [nil | +(random(intruder,
                                  random1)),
                  nil] &
```

The intruder can generate a random value like the `client-hello` and `server-hello` messages do. It can be sent to disrupt the flow of the protocol or to mimic the client or the server.

```
    :: random1 :: [nil | +(scalar(intruder,
                           random1)),
                  nil] &
```

The intruder action, the intruder can take a scalar and send it to disrupt the flow of the protocol.

```
    :: random1 :: [nil | +(session(intruder,
                                   random1)),
                  nil] &
```

The intruder can also generate a session to try to deceive the entity with which it communicates during the message exchange.

```
    :: random1 :: [nil | +(post-quantum-secret-key(intruder,
                                                   random1)),
                  nil] &
```

The intruder can generate its own Post-Quantum secret key in order to try fooling the entity he's communicating with.

```
:: nil :: [nil | -(Master-Secret1),
                 -(Msg1),
                 +(encryption(Master-Secret1,
                              Msg1)),
          nil] &
:: nil :: [nil | -(Master-Secret1),
                 -(Msg1),
                 -(Msg2),
                 -(Msg3),
                 -(Msg4),
                 -(Msg5),
                 +(client-encryption(Master-Secret1,
                                     Msg1,
                                     Msg2,
                                     Msg3,
                                     Msg4,
                                     Msg5)),
          nil] &
:: nil :: [nil | -(Master-Secret1),
                 -(Msg1),
                 -(Msg2),
                 -(Msg3),
                 -(Msg4),
                 -(Msg5),
                 -(Msg6),
                 +(server-encryption(Master-Secret1,
                                     Msg1,
                                     Msg2,
                                     Msg3,
                                     Msg4,
                                     Msg5,
                                     Msg6)),
          nil] &
:: nil :: [nil | -(Master-Secret1),
                 -(Msg1),
                 +(decryption(Master-Secret1,
                              Msg1)),
          nil] &
```

When the intruder intercepts `Master-Secret1` and `Msg1`, he can create and send an `encryption` using these components as shown by the first process. If the intruder the intercepts `Master-Secret1` and `Msg1` through `Msg5`, they can create a `client-encryption` using these components, as shown by the second process. If the intruder intercepts `Master-Secret1` and `Msg1` through `Msg6`, they can create a `server-encryption` using these components, as shown by the third process. Finally, if the intruder intercepts `Master-Secret1` and `Msg1`, they can perform a `decryption` using these components, as shown by the fourth process.

```
:: nil :: [nil | -(exponentiation(p,
                                  Scalar1)),
                 -(exponentiation(p,
                                  Scalar2)),
                 +(exponentiation(p,
                                  Scalar1 * Scalar2)),
           nil] &
```

When the intruder intercepts `exponentiation(p, Scalar1)` and `exponentiation(p, Scalar2)`, they can create `exponentiation(p, Scalar1 * Scalar2)` using these components. This is possible thanks to the capabilities of a quantum computer, which allows the intruder to perform complex calculations that would be infeasible with classical computing methods.

```
    :: nil :: [nil | -(Post-Quantum-Public-KeyC),
                     -(Post-Quantum-Secret-KeyC),
                     +(encapsulation-cipher(Post-Quantum-Public-KeyC,
                                            Post-Quantum-Secret-KeyC)),
              nil] &
    :: nil :: [nil | -(Post-Quantum-Public-KeyC),
                     -(Post-Quantum-Secret-KeyC),
                     +(encapsulation-key(Post-Quantum-Public-KeyC,
                                         Post-Quantum-Secret-KeyC)),
              nil] &
    :: nil :: [nil | -(CipherS),
                     -(Post-Quantum-Secret-KeyC),
                     +(decapsulation(CipherS,
                                     Post-Quantum-Secret-KeyC)),
              nil] &
    :: nil :: [nil | -(Msg1),
                     -(PointC),
                     -(Post-Quantum-Public-KeyC),
                     -(Random1),
                     -(Random2),
                     +(signature(intruder,
                                 PointC,
                                 Post-Quantum-Public-KeyC,
                                 Random1,
                                 Random2)),
              nil]
```

When intercepting `Post-Quantum-Public-KeyC` and `Post-Quantum-Secret-KeyC`, the intruder can create `encapsulation-cipher` and `encapsulation-key` using these components, as shown by the first and second processes. If the intruder intercepts `CipherS` and `Post-Quantum-Secret-KeyC`, they can perform `decapsulation` using these components, as shown by the third process. Finally, when the intruder intercepts `Msg1`, `PointC`, `Post-Quantum-Public-KeyC`, `Random1`, and `Random2`, they can create a `signature` signed by the intruder using these components, as shown by the fourth process.

6.5 Experiments Results

In this section, the experiments designed to verify the security of the protocol will be explained in detail. This verification is fundamental to prove the robustness and reliability of the protocol, ensuring it meets the required security standards. The experiments focus on several critical properties, including the overall protocol behavior, the generation and confidentiality of the Elliptic Curve Diffie-Hellman (ECDH) shared secret key, the security of the Post-Quantum Key Encapsulation Mechanism (PQ KEM) shared secret key, and the effectiveness of authentication mechanisms.

6.5.1. Fairness of Protocol Behavior

In order to verify the correct behavior of the protocol, it is necessary to define an equation, in this case `eq ATTACK-STATE(20)`, where a correct communication will try to start and finish successfully. As a mean to prove it, the equation will be divided in two parts. The first part will include the exchange of the messages of the client's side, and the second part will include the exchange of the messages of the server's side. In the following code the structure of the equation can be seen, where the content of every message is replaced by "..." in order to highlight the importance of the structure of the equation.

```
eq ATTACK-STATE(20) =
    :: random1, random2, random3 ::
    [nil,
    + (client-hello(...)),
    - (server-hello(...)),
    - (server-certificate(...)),
    - (server-key-exchange(...)),
    + (client-key-exchange(...)),
    + (client-finished(...)),
    - (server-finished(...)) | nil] &

    :: random1', random2', random3', random4' ::
    [nil,
    - (client-hello(...)),
    + (server-hello(...)),
    + (server-certificate(...)),
    + (server-key-exchange(...)),
    - (client-key-exchange(...)),
    - (client-finished(...)),
    + (server-finished(...)) | nil]
    || empty
    || nil
    || nil
    || nil
    [nonexec] .
```

At the conclusion of the code execution, the only term that is displayed is "empty". This indicates that there is no specific target or item being sought by the intruder during the exchange of protocol messages. The presence of this term signifies that the equation in question is intended to ensure proper communication according to the protocol's requirements, thus validating the correct functioning of the communication process.

To assess the outcomes of the equation's execution, the "`red summary`" command will be employed. This command is utilized to verify both the number of states that have been reached and the correctness of the solution states. Additionally, "`red initials`" will be used to confirm that the solution states meet the correctness criteria. The following results have been observed and recorded during the execution process:

```
----------------------------------------------------------------------
red summary(20, 0) .
105629002 rewrites in 69124ms cpu (69124ms real)
result: 1 state, 0 solutions
----------------------------------------------------------------------
red summary(20, 1) .
263852128 rewrites in 109896ms cpu (109896ms real)
result: 49 states, 0 solutions
----------------------------------------------------------------------
red summary(20, 2) .
rewrites: 1705140951 rewrites in 704220ms cpu (704220ms real)
result: 83 states, 0 solutions
----------------------------------------------------------------------
red summary(20, 3) .
4089415784 rewrites in 1864704ms cpu (1864704ms real)
result: 139 states, 0 solutions
----------------------------------------------------------------------
red summary(20, 4) .
14029871741 rewrites in 6762356ms cpu (6762355ms real)
result: 250 states, 0 solutions
----------------------------------------------------------------------
red summary(20, 5) .
44591918782 rewrites in 22054556ms cpu (22054556ms real)
result: 419 states, 0 solutions
----------------------------------------------------------------------
red summary(20, 6) .
128016039601 rewrites in 66314740ms cpu (66314790ms real)
result: 698 states, 0 solutions
----------------------------------------------------------------------
red summary(20, 7) .
296549708753 rewrites in 351244040ms cpu (351244222ms real)
result: 760 states, 1 solution
----------------------------------------------------------------------
```

During the execution, it can be seen that a solution has been found in the 7th depth, adding a total of `449123.636` seconds, which is equals to `5.198` days. The solution is found in the following combination of states:

```
< 1 . 33 . 1 . 1 . 2 . 2 . 2 . 1 >
```

In order to verify the correctness of the verification, the found solution must be analyzed using the "`red initials`" command, displayed as a diagram, shown in Figure 6.1:

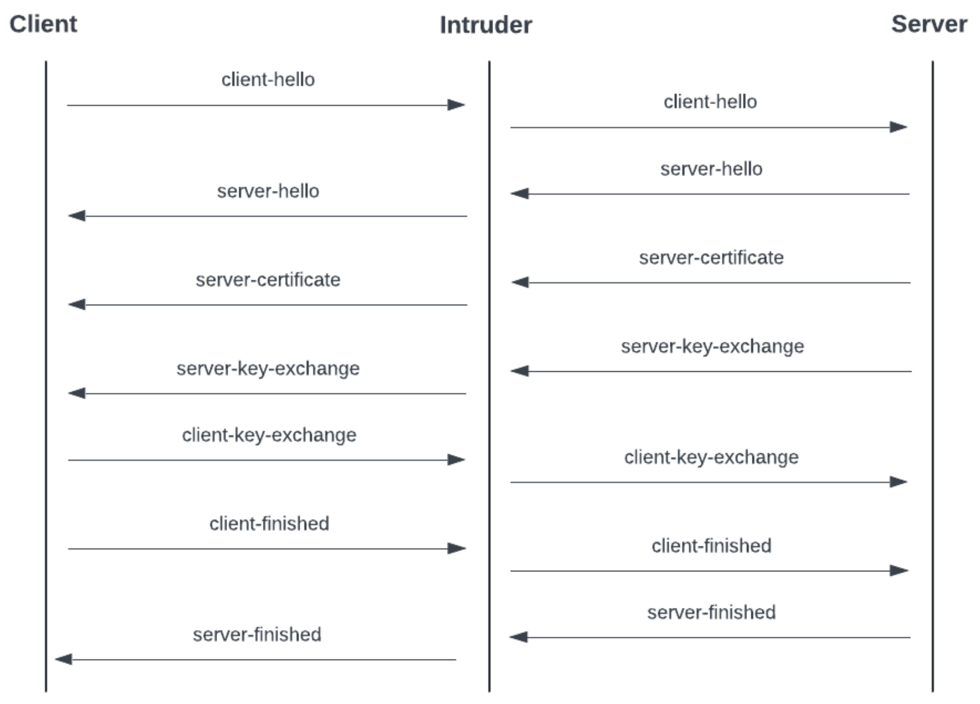


Figure 6.1: Protocol behavior "Attack"

6.5.2. Learning of ECDH Shared Secret Key

With the purpose of verifying the viability of the protocol, two important parts must be taken into account, the ECDH shared secret key and the PQ KEM shared secret key. In this equation the secrecy property of the ECDH shared secret key will be tested. The equation includes one part, where the messages of the client's side will appear.

```
eq ATTACK-STATE(0) =
    :: random1, random2, random3 ::
    [nil,
    + (client-hello(...)),
    - (server-hello(...)),
    - (server-certificate(...)),
    - (server-key-exchange(...)),
    + (client-key-exchange(...)),
    + (client-finished(...)),
    - (server-finished(...)) | nil]
    || exponentiation(
           PointS,
           scalar(
              ClientC,
              random2
           )
       )
     inI, empty
    || nil
    || nil
    || nil
    [nonexec] .
```

At the end of the equation, it has been specified what the intruder is trying to learn during this attack, in this case the ECDH shared secret key, which first appears during the `client-key-exchange` message, which has been represented during the modeling as:

```
exponentiation(
    PointS,
    scalar(
        ClientC,
        random2
    )
)
```

As a result of the execution of the secrecy property ECDH shared secret key equation, the following results have been accomplished:

```
----------------------------------------------------------------------
red summary(0, 0) .
105629054 rewrites in 68512ms cpu (68514ms real)
result: 1 state, 0 solutions
----------------------------------------------------------------------
red summary(0, 1) .
71530603 rewrites in 39784ms cpu (39783ms real)
result: 17 states, 0 solutions
----------------------------------------------------------------------
red summary(0, 2) .
234460360 rewrites in 113248ms cpu (113245ms real)
result: 36 states, 0 solutions
----------------------------------------------------------------------
red summary(0, 3) .
597741091 rewrites in 272424ms cpu (272425ms real)
result: 52 states, 0 solutions
----------------------------------------------------------------------
red summary(0, 4) .
1189833199 rewrites in 519612ms cpu (519613ms real)
result: 57 states, 0 solutions
----------------------------------------------------------------------
red summary(0, 5) .
1635338108 rewrites in 677092ms cpu (677089ms real)
result: 52 states, 0 solutions
----------------------------------------------------------------------
red summary(0, 6) .
2191914230 rewrites in 881000ms cpu (881000ms real)
result: 55 states, 0 solutions
----------------------------------------------------------------------
red summary(0, 7) .
2690802363 rewrites in 1090080ms cpu (1090081ms real)
result: 58 states, 0 solutions
----------------------------------------------------------------------
```

```
red summary(0, 8) .
3229407714 rewrites in 1323264ms cpu (1323262ms real)
result: 71 states, 0 solutions
----------------------------------------------------------------------
red summary(0, 9) .
6322769174 rewrites in 2606004ms cpu (2606003ms real)
result: 125 states, 0 solutions
----------------------------------------------------------------------
red summary(0, 10) .
14973826792 rewrites in 6388292ms cpu (6388291ms real)
result: 182 states, 1 solution
----------------------------------------------------------------------
red summary(0, 11) .
22749583592 rewrites in 10504524ms cpu (10504523ms real)
result: 209 states, 1 solution
----------------------------------------------------------------------
red summary(0, 12) .
31198681045 rewrites in 20038220ms cpu (20038218ms real)
result: 220 states, 1 solution
----------------------------------------------------------------------
red summary(0, 13) .
29998697541 rewrites in 33445988ms cpu (33446036ms real)
result: 176 states, 1 solution
----------------------------------------------------------------------
red summary(0, 14) .
18933212218 rewrites in 33662604ms cpu (33662603ms real)
result: 102 states, 1 solution
----------------------------------------------------------------------
red summary(0, 15) .
7936801126 rewrites in 15562244ms cpu (15562245ms real)
result: 41 states, 1 solution
----------------------------------------------------------------------
red summary(0, 16) .
2012458585 rewrites in 3722976ms cpu (3722974ms real)
result: 9 states, 1 solution
----------------------------------------------------------------------
red summary(0, 17) .
198251789 rewrites in 305760ms cpu (305758ms real)
result: 1 state, 1 solution
----------------------------------------------------------------------
```

During the totality of the execution, which lasted `131221.663` seconds, which is equivalent to `1.519` days, it can be seen that a solution has been found in the 10th depth, up to the 17th depth. This solution is found in the following combination of states:

```
< 1 . 85 . 1 . 22 . 21 . 17{2} . 1{2} . 2 . 2 . 2 . 1 >
```

It can be stated that the when a intruder has the potential abilities to break the ECDH taking the advantages of the Quantum Computers, this side of the protocol is not safe. In order to verify the accuracy of the result, the found solution must be analyzed using the "`red initials`" command, whose result is represented in Figure 6.2, where the black transitions represent the capture and processing of the exponentiation used in the ECDH:

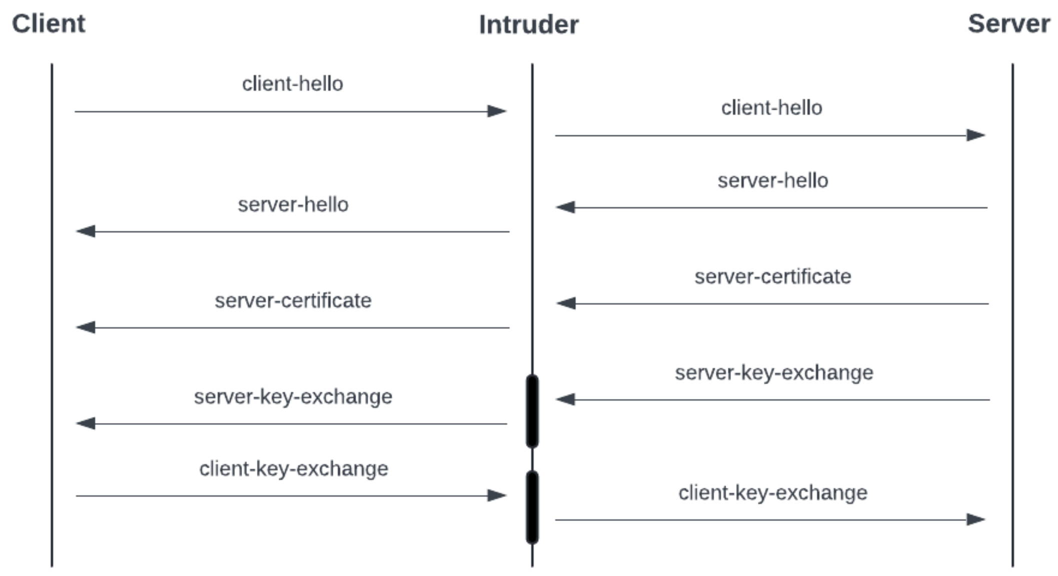


Figure 6.2: ECDH shared secret key Attack

6.5.3. Unable to learn PQ KEM Shared Secret Key

Once the secrecy property of the ECDH shared secret key, the Post-Quantum part must be verified. In the following equation the secrecy property of the PQ KEM shared secret key will be put under test. The equation is formed by one part, where the messages of the client's side will appear.

```
eq ATTACK-STATE(1) =
    :: random1, random2, random3 ::
    [nil,
    + (client-hello(...)),
    - (server-hello(...)),
    - (server-certificate(...)),
    - (server-key-exchange(...)),
    + (client-key-exchange(...)),
    + (client-finished(...)),
    - (server-finished(...)) | nil]
    || encapsulation-key(
           Post-Quantum-Public-KeyS,
           post-quantum-secret-key(
               ClientC,
               random3
           )
       )
     inI, empty
    || nil
    || nil
    || nil
    [nonexec] .
```

At the end of the equation appears the part of the message exchange that the intruder is trying to intercept, in this case the PQ KEM shared secret key, making the first appearance in the `client-finished` message. Even if the process of the generation of the PQ KEM shared secret key does not start in the `client-finished` message, but in the `client-key-exchange` message, the intruder is trying to learn the shared secret key, not how it has been created.

```
encapsulation-key(
    Post-Quantum-Public-KeyS,
    post-quantum-secret-key(
        ClientC,
        random3
    )
)
```

As a result of the execution of the secrecy property PQ KEM shared secret key equation, the following results have been accomplished:

```
----------------------------------------------------------------------
red summary(1, 0) .
91440886 rewrites in 60096ms cpu (60096ms real)
result: 1 state, 0 solutions
----------------------------------------------------------------------
red summary(1, 1) .
53727582 rewrites in 28428ms cpu (28427ms real)
result: 15 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 2) .
180465530 rewrites in 81136ms cpu (81134ms real)
result: 26 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 3) .
437332677 rewrites in 183856ms cpu (183854ms real)
result: 31 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 4) .
821185247 rewrites in 325536ms cpu (325537ms real)
result: 35 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 5) .
1231084640 rewrites in 480564ms cpu (480564ms real)
result: 37 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 6) .
1234500079 rewrites in 493804ms cpu (493803ms real)
result: 29 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 7) .
765173182 rewrites in 321512ms cpu (321511ms real)
result: 20 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 8) .
```

```
1227367176 rewrites in 491988ms cpu (491987ms real)
result: 45 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 9) .
4649827603 rewrites in 1849616ms cpu (1849616ms real)
result: 114 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 10) .
16041962248 rewrites in 6600400ms cpu (6600400ms real)
result: 225 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 11) .
41280159809 rewrites in 18328008ms cpu (18328008ms real)
result: 353 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 12) .
75615704479 rewrites in 57355092ms cpu (57355138ms real)
result: 436 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 13) .
93432928429 rewrites in 258495868ms cpu (258496013ms real)
result: 414 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 14) .
76613352796 rewrites in 376189924ms cpu (376190107ms real)
result: 297 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 15) .
41245770284 rewrites in 246813584ms cpu (246813725ms real)
result: 155 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 16) .
14549422090 rewrites in 87836592ms cpu (87836637ms real)
result: 53 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 17) .
3070237068 rewrites in 21114204ms cpu (21114205ms real)
result: 9 states, 0 solutions
----------------------------------------------------------------------
red summary(1, 18) .
235956981 rewrites in 2384920ms cpu (2384919ms real)
result: 0 states, 0 solutions
----------------------------------------------------------------------
```

The verification results demonstrate the robustness of the protocol designed to ensure the secrecy of the Post-Quantum Key Encapsulation Method (PQ KEM) shared secret key. All tests resulted in 0 solutions, meaning no counterexamples or violations of the secrecy property were discovered. It took a total time of `1079435.681` seconds, which is equals to `12.493` days.

This extensive testing, which involved examining numerous states and performing billions of rewrites, consistently showed that the protocol maintained the secrecy of the shared secret key in all scenarios tested. The absence of any detected security breaches or weaknesses across these rigorous tests provides strong evidence that the secrecy property of the PQ KEM shared secret key is secure. Therefore, we can confidently conclude that the protocol ensures the confidentiality of the PQ KEM shared secret key against the analyzed threats and attack vectors.

6.5.4. Authentication Protection

The authentication property tries to verify a situation where the client has apparently completed the handshake with the server, by sending the `client-finished` message, and it receives a valid `server-finished` message, but actually it has never been executed such a server instance with the client.

```
eq ATTACK-STATE(2) =
    :: random1, random2, random3 ::
    [nil,
    + (client-hello(...)),
    - (server-hello(...)),
    - (server-certificate(...)),
    - (server-key-exchange(...)),
    + (client-key-exchange(...)),
    + (client-finished(...)),
    - (server-finished(...)) | nil]
    || empty
    || nil
    || nil
    || never(
       :: random1', random2', random3', random4' ::
       [nil |
       - (client-hello(...)),
       + (server-hello(...)),
       + (server-certificate(...)),
       + (server-key-exchange(...)),
       - (client-key-exchange(...)),
       - (client-finished(...)),
       + (server-finished(...)), nil ]
       & Strand-Set:StrandSet || Intruder-Knowledge:IntruderKnowledge
       )
    [nonexec] .
```

The equation is composed by two parts. The first part includes the client's side as seen in the other attacks. The second part is formed by the word never, which includes the client's side code, changing the signs that indicate the sending and receiving messages to simulate the server behavior. The following results have been obtained:

```
----------------------------------------------------------------------
red summary(2, 0) .
105629002 rewrites in 65332ms cpu (65331ms real)
result: 1 state, 0 solutions
----------------------------------------------------------------------
red summary(2, 1) .
32709935 rewrites in 18764ms cpu (18761ms real)
```

```
result: 8 states, 0 solutions
----------------------------------------------------------------------
red summary(2, 2) .
32405382 rewrites in 18280ms cpu (18281ms real)
result: 4 states, 0 solutions
----------------------------------------------------------------------
red summary(2, 3) .
7718093 rewrites in 4596ms cpu (4597ms real)
result: 0 states, 0 solutions
----------------------------------------------------------------------
```

The verification results provided for the attack on the authentication property reveal that the protocol remains resilient against authentication threats. The tests, encompassing numerous states and millions of rewrites, consistently resulted in 0 solutions, indicating no counterexamples or violations of the authentication property were found. It took a total time of `106.972` seconds, which is equals to `0.001` days.

The detailed analysis showed that even under different scenarios and attack vectors, the protocol maintained its integrity and successfully authenticated the involved parties. This thorough testing process, which involved significant computational resources and extensive state exploration, reinforces the robustness of the protocol's authentication mechanisms. Therefore, we can conclusively state that the protocol effectively ensures the authentication property, providing strong evidence that it can withstand potential attacks aimed at compromising this aspect of security.

6.6 Experiment Conclusions and Comparative

In this section, a comprehensive analysis of the experimental results obtained from the proposed Hybrid Post-Quantum TLS protocol is presented. The performance, security, and overall effectiveness are evaluated by comparing it with the model [10] introduced by Duong Dinh Tran, Canh Minh Do, Santiago Escobar, and Kazuhiro Ogata. This comparative study is crucial for highlighting the strengths and potential improvements of the proposed protocol in the context of post-quantum cryptographic security.

The team of Duong Dinh Tran, Canh Minh Do, Santiago Escobar, and Kazuhiro Ogata utilized a supercomputer for their tests, equipped with 1.5 TB of memory and four 2.8 GHz microprocessors, each featuring 16 cores. In contrast, the tests for the proposed Hybrid Post-Quantum TLS protocol were conducted on a computer with 512 GB of memory and two Intel(R) Xeon(R) Silver 4215R CPUs at 3.20 GHz, each having 8 cores and 2 threads per core. In the following table, the letter 'H' means that the experiments are still running at that moment, meaning 'Handling', while the letter 'F' means that the experiments has finished, referring to 'Finished'.

Attack number	**Previous version [10]**			**This thesis**		
	Time (hours)	**Depth**	**Result**	**Time (hours)**	**Depth**	**Result**
0	1722.34	H 11	∅	36.46	F 17	X
1	1722.34	H 11	∅	299.83	F 18	∅
2	1722.34	H 18	∅	0.03	F 03	∅

Table 6.1: Attacks results comparison

The comparative analysis reveals several significant differences between the previous and new versions. Notably, the new version of the protocol exhibits a substantial reduction in time required to perform attacks. For instance, attack number 0 shows a decrease from 1722.34 hours in the previous version to 36.46 hours in the new version. This improvement is complemented by an increase in the depth from H 11 to F 17, indicating enhanced complexity in handling the attack, and resulting in a successful outcome, denoted by X. Here's a further look to the total states generated:

Depth	**Previous Attack 0**	**New Attack 0**
1	15	17
2	30	36
3	51	52
4	86	57
5	150	52
6	261	55
7	435	58
8	695	71
9	1067	125
10	1597	182
11	2374	209
12	3679	220
13	-	176
14	-	102
15	-	41
16	-	9
17	-	1
Total	**10440**	**1463**

Table 6.2: Generated states for Attack 0

The comparison shows that the new version of the protocol generates significantly fewer states (1463) compared to the previous version (10440). This substantial reduction in the number of states generated at each depth indicates a more efficient protocol in terms of state space exploration, leading to faster and more resource-efficient attack processing.

Similarly, attack number 1 demonstrates a reduced time from 1722.34 hours to 299.83 hours. Although the result remains the same ($\varnothing$), the depth is increased from H 11 to F 18, suggesting improved resistance to the attack.

These findings underscore the enhanced efficiency and security of the proposed Hybrid Post-Quantum TLS protocol compared to the previous version. The reduction in processing time and the increased depth in attack handling are indicative of significant advancements in the protocol's robustness and overall effectiveness.

Furthermore, a detailed comparison of the total amount of states generated at each depth for Attack 1 between the previous and new versions of the protocol will be provided. This comparison aims to highlight the improvements in efficiency and performance of the new protocol, emphasizing the enhanced state space exploration capabilities and overall effectiveness in handling cryptographic attacks:

Depth	Previous Attack 1	New Attack 1
1	15	15
2	30	26
3	56	31
4	101	35
5	165	37
6	262	29
7	408	20
8	610	45
9	871	114
10	1216	225
11	1757	353
12	2634	436
13	-	414
14	-	297
15	-	155
16	-	53
17	-	9
18	-	0
Total	**8125**	**2294**

Table 6.3: Generated states for Attack 1

The comparison highlights that the enhanced version of the protocol can go deeper than the original version. There is also a big reduction of the number of states generated, where the new protocol (2294) generates less states that the original version (8125). This reduction in the number of states generated at each depth. There is also a big difference in the execution time, where we can compare the original execution time of `1722.34` hours to the `0.3` hours in the new protocol.

The verification results provided for the attack on the authentication property reveal notable improvements in the new protocol. The tests show a significant reduction in time required to verify the authentication, demonstrating enhanced efficiency. For instance, the total time for verification was reduced to `106.972`, which is `0.03` hours. Despite the extensive testing involving millions of rewrites and numerous states, the protocol consistently yielded 0 solutions, confirming the absence of any authentication breaches. These results underscore the protocol's robustness in maintaining the integrity and authenticity of the communication, showcasing its enhanced resistance to authentication attacks:

Depth	Previous Attack 2	New Attack 2
1	8	8
2	5	4
3	2	0
4	2	-
5	2	-
6	2	-
7	2	-
8	2	-
9	2	-
10	4	-
11	10	-
12	19	-
13	40	-
14	95	-
15	246	-
16	602	-
17	1355	-
Total	**2398**	**12**

Table 6.4: Generated states for Attack 2

The comparison reveals that the new version of the protocol is much more efficient in terms of state generation. The new protocol produces only 12 states in total, a stark contrast to the 2398 states generated by the previous version. This dramatic decrease in the number of states at each depth underscores the improved efficiency of the new protocol's state space exploration. Furthermore, the new protocol achieves the desired outcome by reaching only a depth of 3, whereas the previous version required going up to a depth of 17. This efficiency in both state generation and depth exploration highlights the significant advancements in the new protocol's performance, resulting in more rapid and resource-efficient attack processing.

CHAPTER 7

Conclusion

In this thesis, a formal verification of the Hybrid Post-Quantum TLS protocol was conducted, comparing a new variant with an existing implementation. The analysis focused on verifying key security properties, including fairness of protocol behavior, learning of the ECDH shared secret key, inability to learn the PQ KEM shared secret key, and authentication protection.

To achieve these results, significant groundwork was necessary. This included a thorough study of the TLS RFC [3], the IETF drafts on Hybrid TLS [4], a deep understanding of the specifications provided by Duong Dinh Tran, Canh Minh Do, Santiago Escobar and Kazuhiro Ogata [10], the codification of the protocol and performing the analyzes. These efforts were crucial in identifying vulnerabilities that could potentially compromise security under certain conditions. This thesis plays a crucial role in the ongoing transition to quantum-resistant cryptography, providing a secure foundation for future communications.

The contribution to the field of post-quantum cryptography includes offering a formally verified protocol. The verification results have been made publicly available for further research and validation.

For additional details, the code and verification models can be accessed on GitHub.

Bibliography

[1] Santiago Escobar, Catherine Meadows, José Meseguer, *The Maude NPA Manual (Version 3.1)*, University of Illinois at Urbana-Champaign, 2011. https://maude.cs.illinois.edu/w/images/9/90/Maude-NPA_manual_v3_1.pdf

[2] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, Carolyn Talcott, *Maude Manual, Version 3.0*, University of Illinois at Urbana-Champaign, 2022. https://maude.lcc.uma.es/maude-manual/maude-manualch3.html#x15-280003.3

[3] T. Dierks, E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC Editor, RFC 5246, 2008. https://www.rfc-editor.org/rfc/rfc5246

[4] M. Campagna, E. Crockett, *Hybrid Post-Quantum Key Encapsulation Methods (PQ KEM) for Transport Layer Security 1.2 (TLS)*, Internet-Draft, IETF, 2023. https://datatracker.ietf.org/doc/html/draft-campagna-tls-bike-sike-hybrid

[5] Andreas Mars, *Elliptic Curves*, 2006. https://www.maths.tcd.ie/pub/Maths/Courseware/499/2006/Mars/ellcurves.pdf

[6] BIKE Project, *BIKE: Bit Flipping Key Encapsulation*. https://bikesuite.org/

[7] CRYSTALS-Kyber Project, *KYBER: Cryptographic Suite for Algebraic Lattices*. https://pq-crystals.org/kyber/

[8] SIKE Project, *Supersingular Isogeny Key Encapsulation*. https://sike.org/

[9] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, *BIKE: Bit Flipping Key Encapsulation*, 2017. https://hal.science/hal-01671903/document

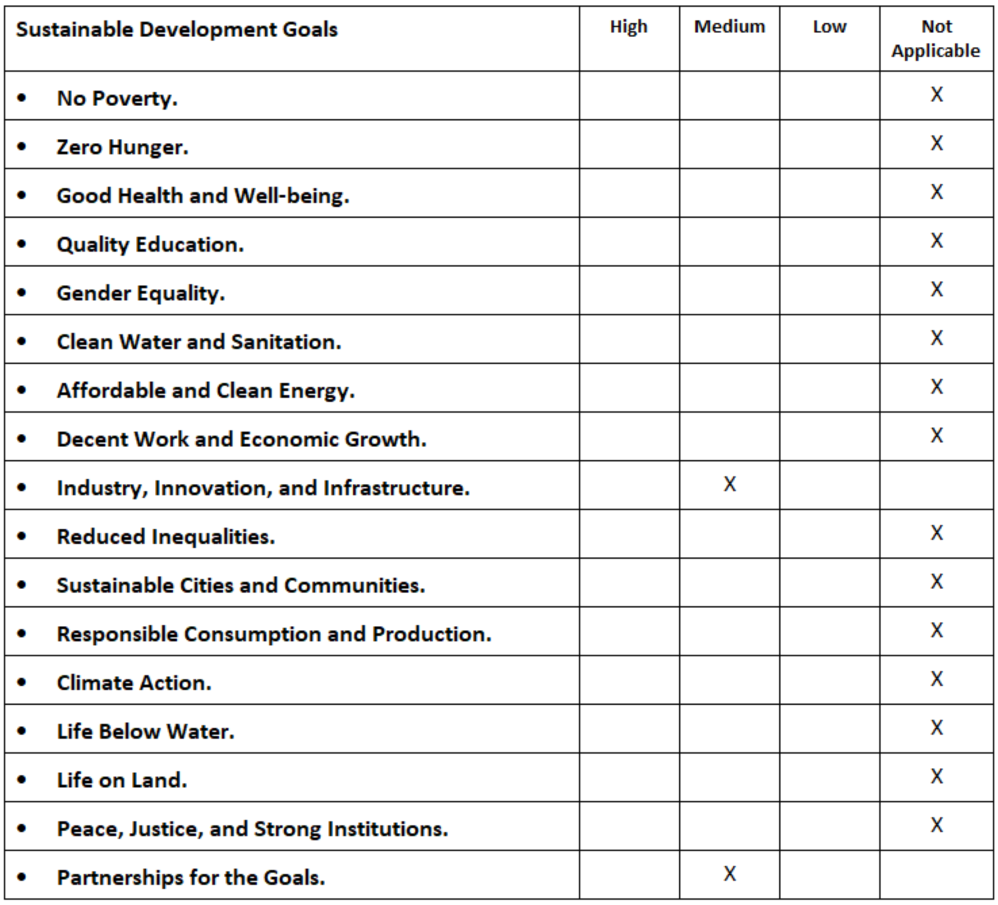
[10] Duong Dinh Tran, Canh Minh Do, Santiago Escobar, Kazuhiro Ogata, *Hybrid post-quantum Transport Layer Security formal analysis in Maude-NPA and its parallel version*, *PeerJ Computer Science*, vol. 1, p. e1556, 2015. https://peerj.com/articles/cs-1556/

[11] Y. Nir, S. Josefsson, M. Pegourie-Gonnard *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier*, RFC Editor, RFC 8422, July 2018. https://www.rfc-editor.org/info/rfc8422

[12] Shor, Peter W *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*, 1997 http://dx.doi.org/10.1137/S0097539795293172

APPENDIX A

Sustainable Development Goals



Sustainable Development Goals	High	Medium	Low	Not Applicable
• No Poverty.				X
• Zero Hunger.				X
• Good Health and Well-being.				X
• Quality Education.				X
• Gender Equality.				X
• Clean Water and Sanitation.				X
• Affordable and Clean Energy.				X
• Decent Work and Economic Growth.				X
• Industry, Innovation, and Infrastructure.		X		
• Reduced Inequalities.				X
• Sustainable Cities and Communities.				X
• Responsible Consumption and Production.				X
• Climate Action.				X
• Life Below Water.				X
• Life on Land.				X
• Peace, Justice, and Strong Institutions.				X
• Partnerships for the Goals.		X		

Figure A.1: Substainable Development Goals Table

The following sections detail the specific impacts of these advancements on the relevant SDGs, illustrating how they contribute to the broader goals of sustainable development and global cooperation.

- **Industry, Innovation, and Infrastructure (SDG 9)**:
  - **Enhancing Cybersecurity Infrastructure**: improvements in the Hybrid Post-Quantum TLS protocol contribute to the strengthening of cybersecurity infrastructure. By advancing the protocol's robustness against quantum threats, the development supports the creation of more secure communication systems, which is vital for industry-wide cybersecurity advancements.

  - **Promoting Technological Innovation**: significant advancements in protocol modeling demonstrate progress in the field of cryptography. Innovations in this domain foster the development of new technologies and methods, thereby contributing to the advancement of industry infrastructure and stimulating further research and development in cybersecurity.

- **Partnership for the Goals (SDG 17)**:

  - **Facilitating Collaborative Research Efforts**: building upon existing scientific research and refining methodologies contribute to a collaborative advancement in cybersecurity. Such efforts support the formation of partnerships within the global research community, enhancing collective knowledge and addressing shared challenges in the field.
  - **Supporting the Development of Global Standards**: achievements in improving post-quantum cryptographic protocols serve as a basis for standards' development. Dissemination of these findings aids in establishing universally accepted practices, thereby strengthening partnerships across different sectors and contributing to global efforts in enhancing cybersecurity.