PQ TESLA and its Application to DTLS

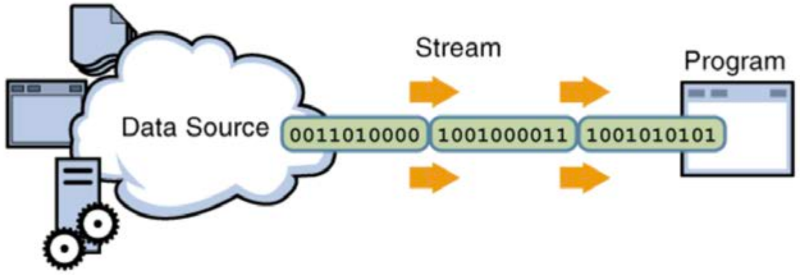
Simpy Parveen

Today's Talk

- Motivation
- Contribution
- Background
  - TESLA Protocol
  - DTLS protocol
  - K2SN-MSS
- TESLA Protocol
  - Sketch of TESLA Protocol
  - Pre-requisites – Time Synchronization, One way key chain, TESLA initialization
  - Message Transmission from sender to receiver
  - Message Authentication at receiver
- DTLS
  - Security Guarantees
- K2SN-MSS
  - Merkle Tree Constructions X
  - Signature Algorithms
- PQ TESLA
  - PQ-TESLA and its Application to DTLS
  - DTLS with Source authentication and Data integrity
  - TESLA Initialization in DTLS HS Layer
  - TESLA Extension in DTLS Record Layer
- Sketch of Implementation
- Function Flow TinyDTLS
- Experimental Setup
- Experimental results -1

Motivation

- Data Stream : A data stream is a sequence of digitally encoded coherent signals.
- Packets : A packet consists of control information(headers, seq no, etc) and user data(payload information).



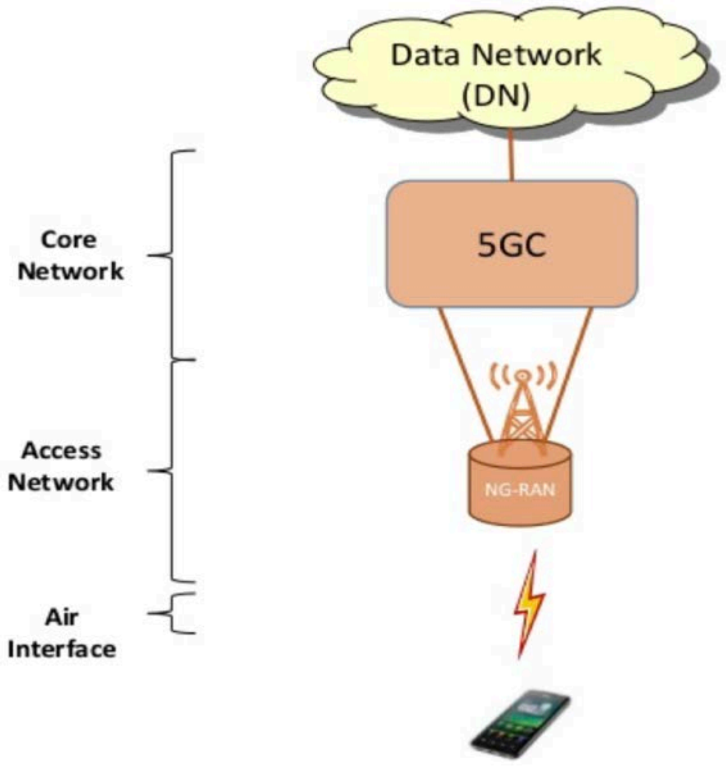
Streaming Media needs :

* Timeliness of data
* Does not need retransmission

➢ Use of UDP Transport protocols – resilient to packet drops

- Popular applications like CoAP and WebRTC use DTLS
- Communication between RAN and Core Network in 5G Network –requires :
    1. Packet Authenticity
    2. Packet Integrity
- Transition to Quantum safe Cryptography

DTLS does not provide PQ security !!

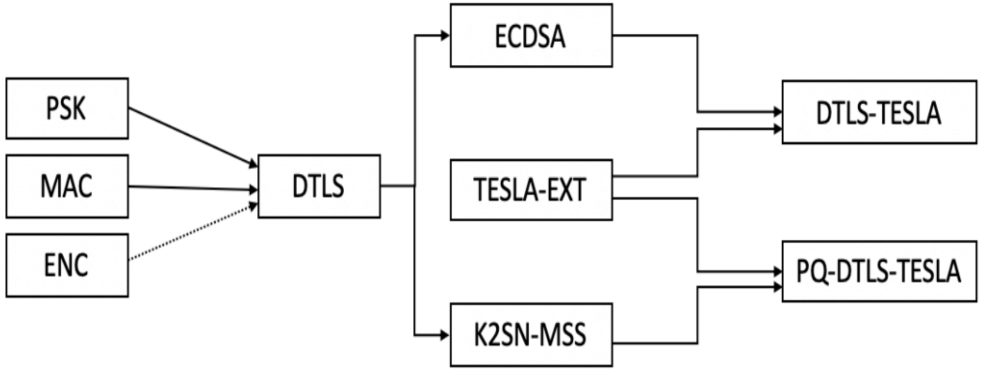


Contributions

Our contributions is two-fold which includes design and implementation of :

1. **PQ TESLA** for the authentication/integrity of the packets generated by the session's sender.
   a. Implement TESLA algorithm 4.
   b. Design & Implement PQ-TESLA

2. **PQ-DTLS** with source authentication and integrity only
   a. Use TinyDTLS library for DTLS
   b. Incorporate PQ-TESLA to DTLS library



3. Performance Evaluation : *Overhead of adding PQ Security*
   a. Comparison :
      - ***DTLS*** : PSK, ENC, MAC
      - ***DTLS-TESLA*** : PSK, ENC(optional), MAC, ECDSA and TESLA-EXT
      - ***PQ-DTLS-TESLA*** : PSK, ENC(optional), MAC, K2SN-MSS and TESLA-EXT

Background

TESLA Protocol

Timed Efficient Stream Loss-tolerant Authentication

DTLS

Datagram Transport Layer Security

K2SN-MSS

K2SN Multi-message Signature Scheme

Sketch of TESLA protocol

Goal : Provides **source authentication and integrity** to secure data stream on per-packet basis.
Idea : *TESLA uses a new MAC key for each packet, which will be sent by the sender after sufficient delay.*

- **Threat Model**
  - The adversary with full control over the network. The adversary can eavesdrop, capture, drop, resend, delay, and alter packets.
  - The adversary's computational resources may be very large, but not unbounded.
- **Participants** :
  - Sender
  - Receiver
- **Security Guarantee**
  - The receiver does not accept any message M_i unless M_i was actually sent by the sender and was not tampered on the way.
- **Cryptographic Primitives**
  - Message Authentication Code(MAC)
  - One-Way Hash Function
  - Digital Signature Scheme

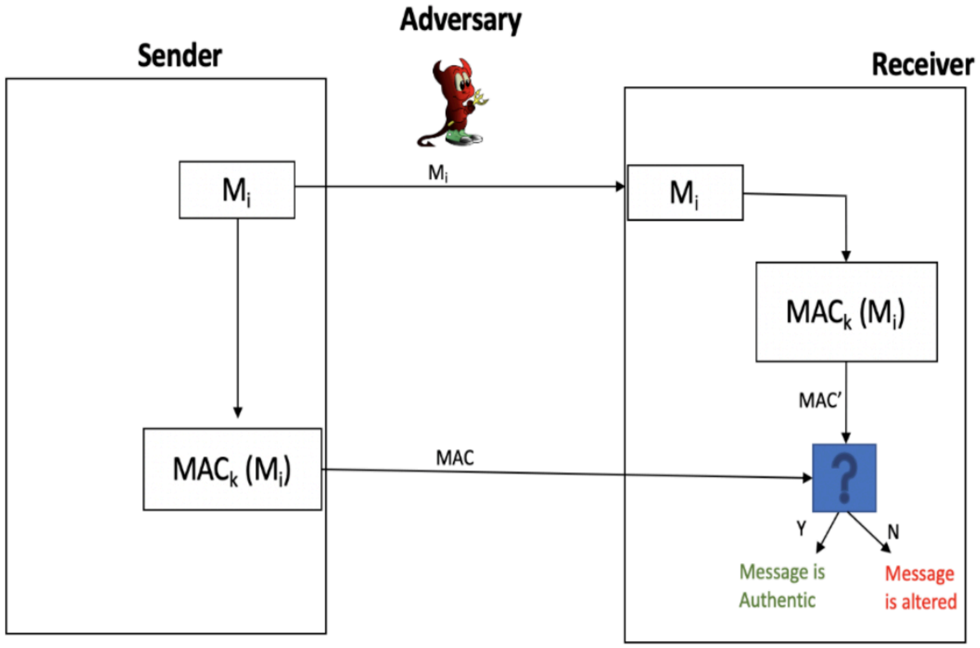


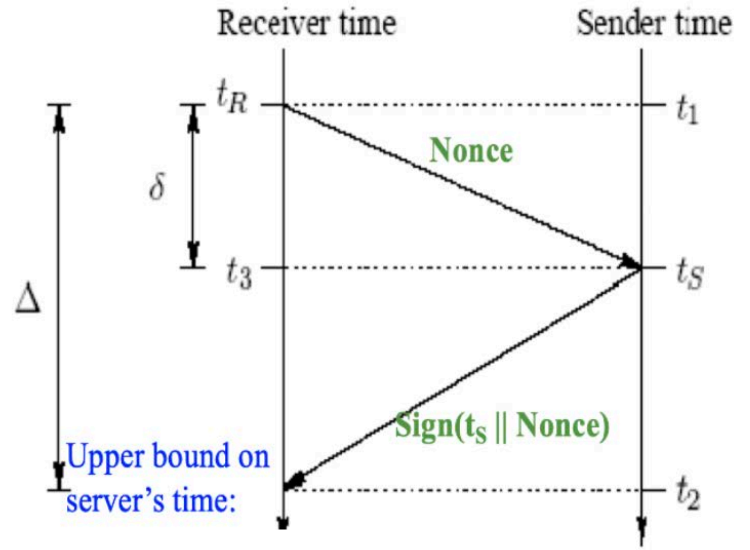
Figure 3.1: A sketch of the TESLA protocol.

Pre-requisite for TESLA

- Time Synchronization
- One-way Key Chain
- TESLA Initialization

Time Synchronization

Goal: Know upper bound on sender's clock



$t_s \le \ t_r \ \ - \ t_R \ \ \pm \ t_S$

t_{S} : Sender's local time when Synchronization request received.

t_s : Sender's local time when Synchronization response is received.

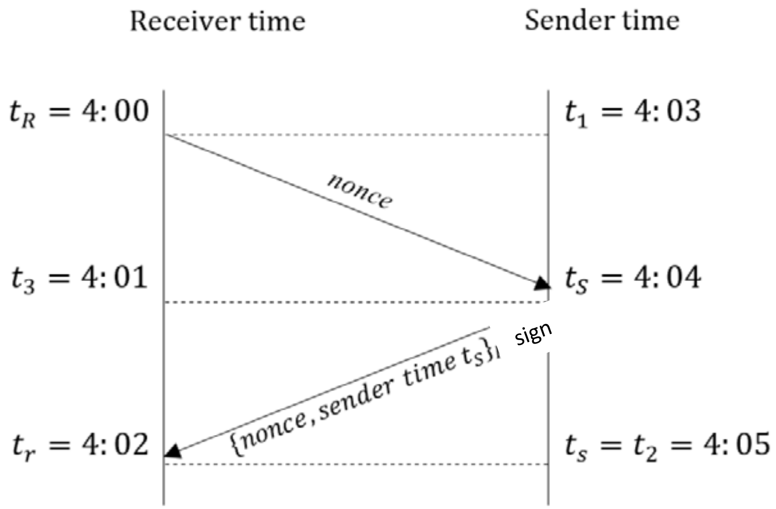
t_R : Receiver's local time Synchronization request packet is sent.

t_r : Receiver's local time when Synchronization response is received.

Δ = Maximum time synchronization error

δ = Exact time difference

Example.



$t_s \le \ t_r \ \ - \ t_R \ \ \pm \ t_S$

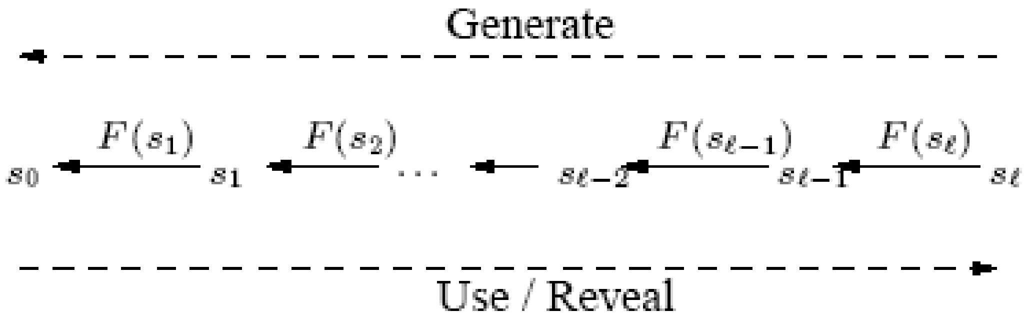
$t_s \le 4{:}\,02 - 4{:}\,00 + 4{:}\,04 = 4{:}\,06$

One-way Key Chain

On-way key chain : The keys are revealed in **reverse** to their generation order:

Generation: S_{last}, S_{last-1}, S_{last-2},…, S_0

Usage(Revealed): S_0, S_1,… S_{last}



- The first element in the chain, is committed to the entire chain: $F^i(s_i) = s_0$
- We can verify that an element s_j is a part of the chain by checking that $F^{j-i}(s_j) = s_i$ for some element s_i that is in our chain (and i< j)
  - S_i commits to S_j if (I < j) and both belong to the chain

TESLA Initialization

- The receiver sends a synchronization request(nonce) and the Sender prepares a synchronization response packet, signed using sender's private-key.

$$
\begin{aligned}
R \rightarrow S &: \text{Nonce} \\
S \rightarrow R &: \{\text{Sender time } t_S, \text{Nonce}, \text{ Interval Rate}, \\
&\quad \text{Interval Id}, \text{ Interval start time}, \\
&\quad \text{Interval key}, \text{Disclosure Lag}\}_{K_S^{-1}}
\end{aligned}
$$

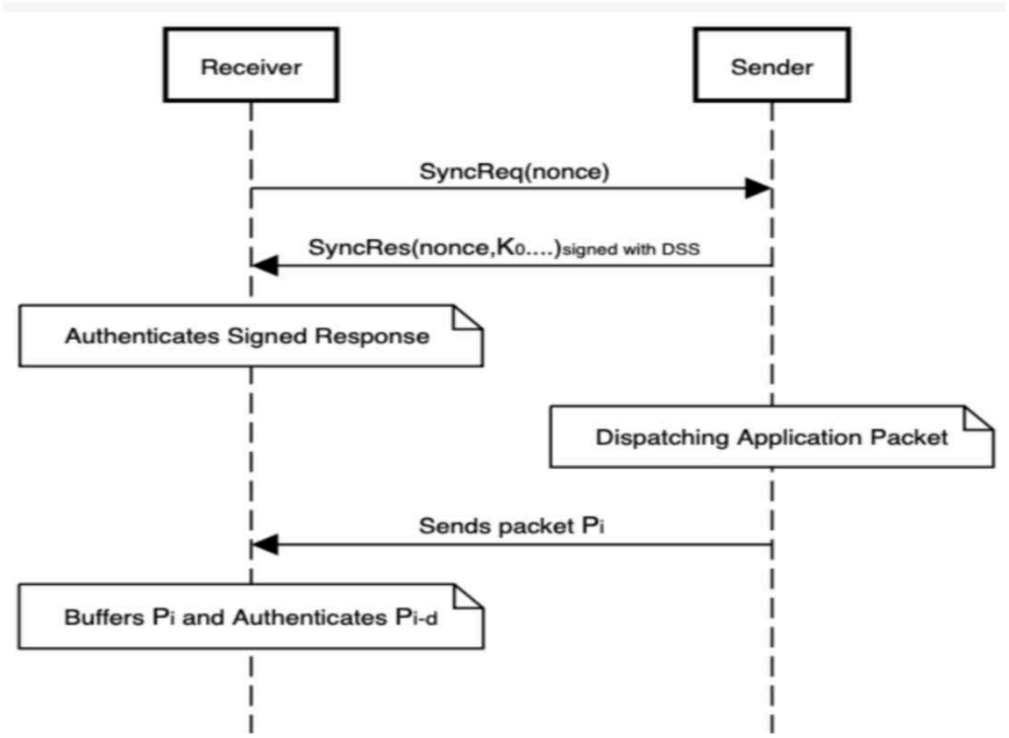


Figure 2.3: Sender-Receiver Operations in TESLA

Message Transmission from TESLA Source to Receiver

Algorithm 1: Basic Scheme
Algorithm 2: Tolerating Packet Loss is achieved using keychain
Algorithm 3: Achieving Fast Transfer Rates by introducing delay parameter*(d)*.

Security condition: A data packet P_i arrived *safely*, if the receiver can unambiguously decide, based on its synchronized time and δ_t, that the sender did not yet send out the corresponding key disclosure packet P_j.

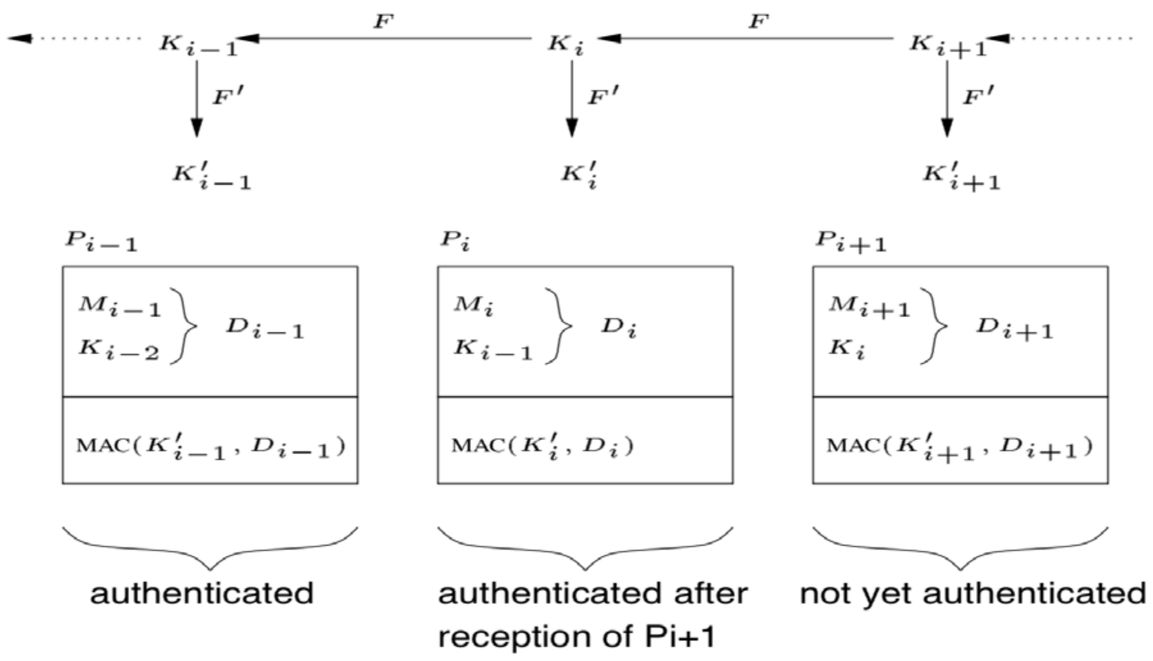
The sender sends the messages after initial synchronization is complete.

- Authentication Tag:

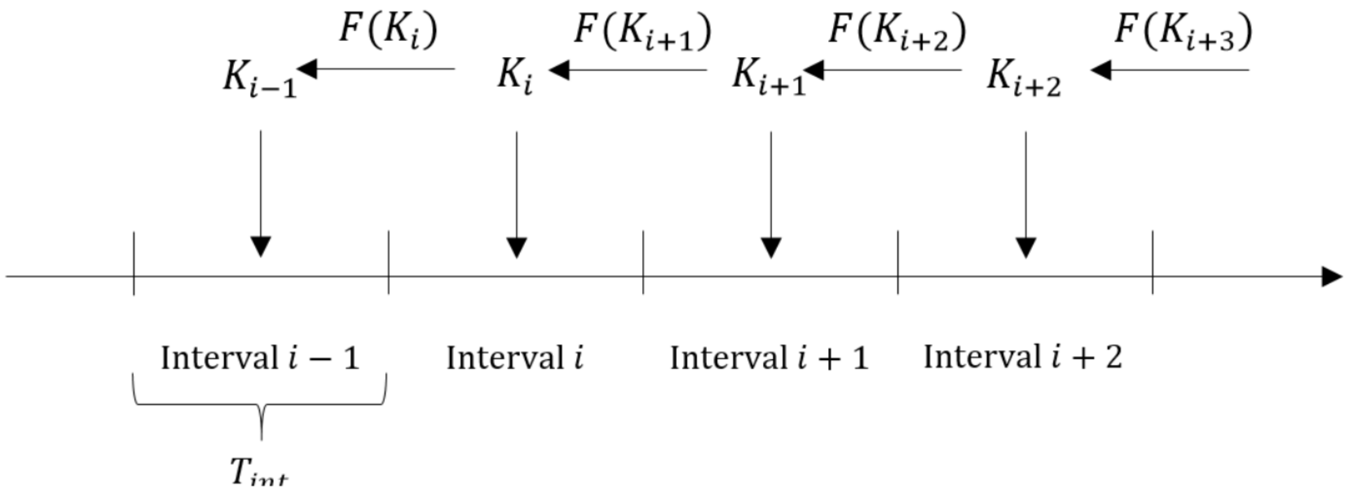
$$(i, HMAC(K_i, M_i), K_{i-d})$$

- To broadcast message M_j in interval i the sender constructs packet as :

$$P_j = \{M_j \| i \| K_{i-d} \| MAC(K_i', M_j)\}$$



Message Authentication at TESLA Receiver



Packet Safety :

- Packet is **SAFE**, if **$x < i + d$,** where $x < [(t_s – T_0) /T_{int}]$ (where t_s is the upper bound on current server's time)

New Index Key Test :

- When current interval is **i** the disclosed key index should be **K_{i-d}**.

Key Verification Test :

- The key revealed in current packet, that is, **K_i** is part of key-chain commitment(**K_0**).

Message Authentication :

- **MAC** verification of previously buffered packet using the revealed key in current packet.

DTLS

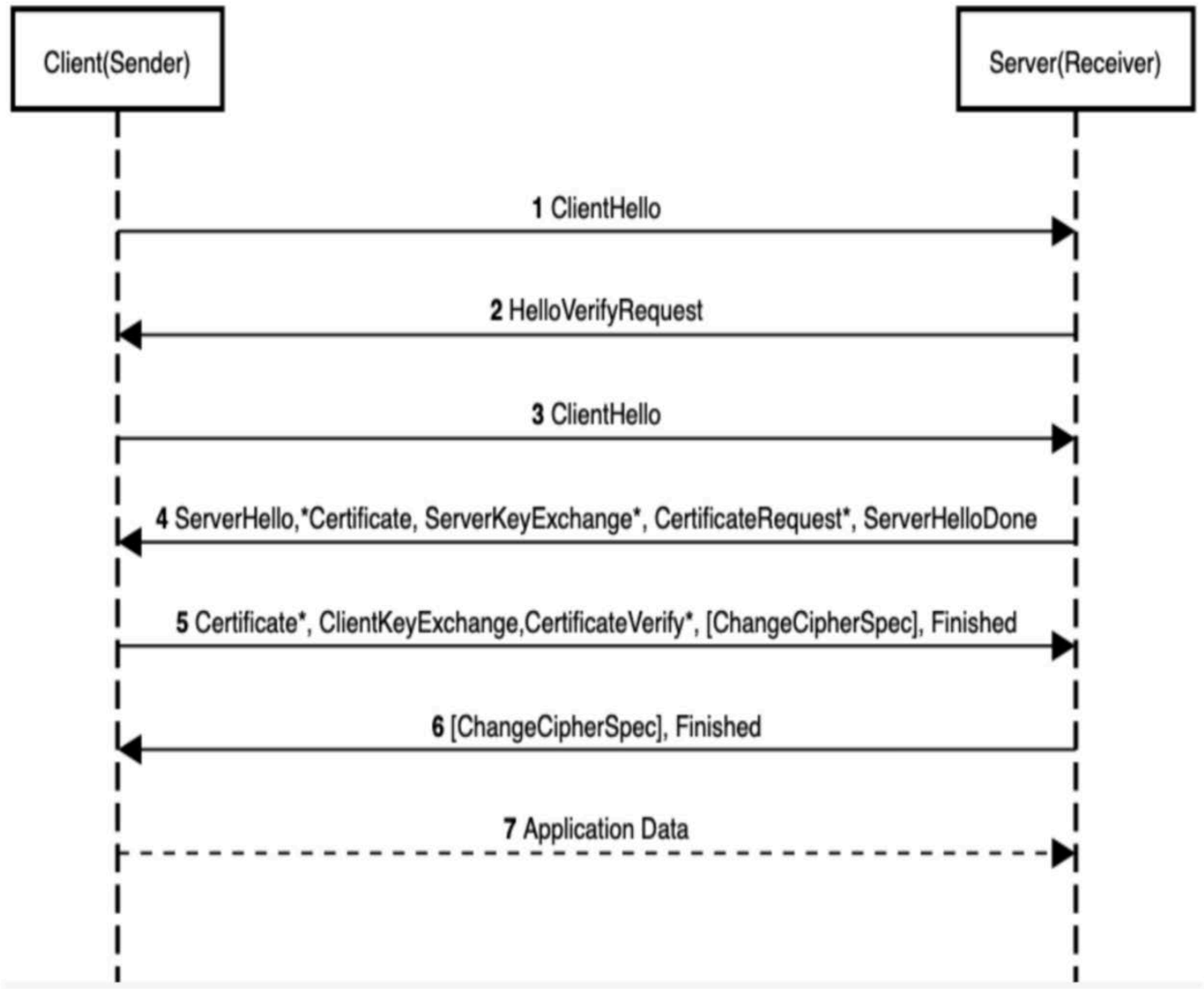


Figure 2.6: DTLS Fully Authenticated Handshake

- The DTLS protocol is designed to secure data between communicating applications.
- Security Guarantees
  - Origin Authentication : Using certificates or Public key cryptography.
  - Confidentiality : Using encryption
  - Integrity : Using HMAC

- DTLS provides data stream authentication for applications built on User Datagram Protocol(UDP) channel.
- DTLS connection has two main phases:
  - DTLS Handshake Protocol
    - Key Exchange
    - Peer Authentication
    - Negotiate Ciphersuite
  - Record Layer Protocol
    - Records are protected with keys exchange during handshake.

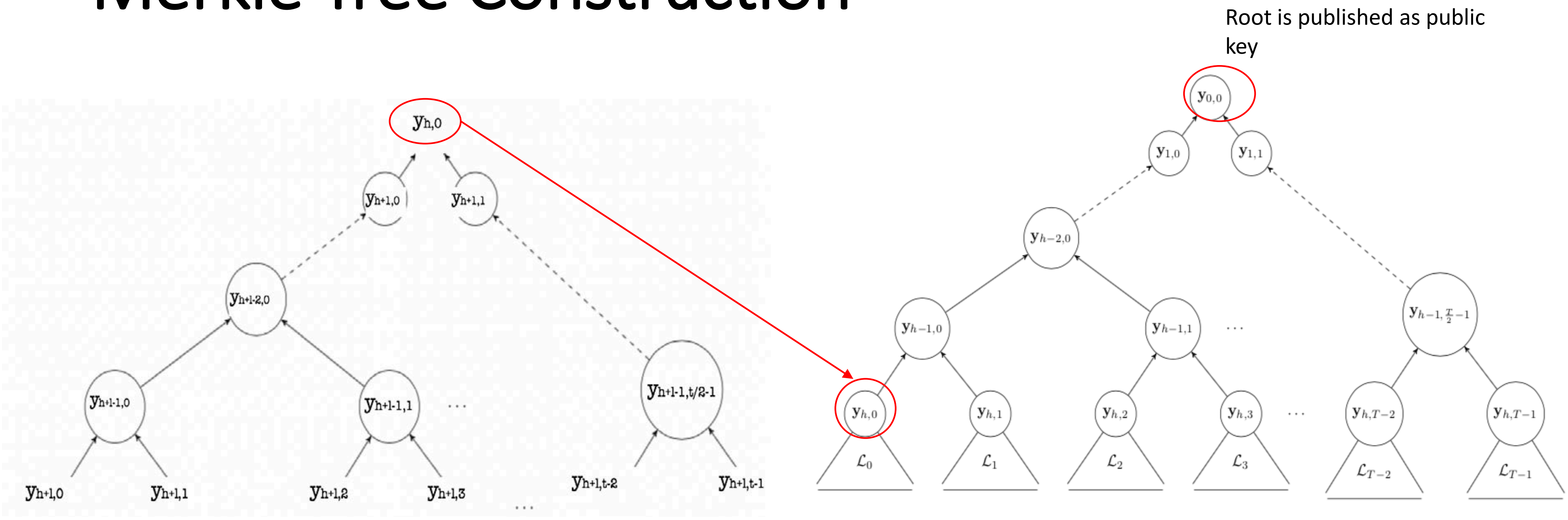
Security Guarantees by DTLS

- **Replay attacks**: The use of explicit sequence number in DTLS's record layer helps mitigate replay attacks.
- **Denial of Service (DoS) attacks**: DTLS makes Denial of Service (DoS) attacks less effective by disabling fragmentation. During the handshake, a stateless cookie exchange prevents DoS attacks like resource consumption attacks and amplification attacks.
- **Handling Invalid Records**: Unlike TLS, DTLS is resilient in the face of invalid records (e.g., invalid formatting, length, MAC, etc.). In general, invalid records SHOULD be silently discarded, thus preserving the association.

K2SN Signature scheme

- K2SN-MSS extends the KSN-OTS to multi-message signature scheme and uses SWIFFT as the underlying hash function.
- Each of KSN-OTS from K2SN-MSS is used to sign a single message, i.e., 2^h KSN-OTS can be generated for signing 2^h messages.
- The parameters of SWIFFT are chosen such that it provides 512-bit classical (256-bit quantum) security for K2SN-MSS against existential unforgeability in chosen message attack (EUF-CMA).
- K2SN-MSS Signature consists of three algorithms:
  - Key Generation
    - Uses Chacha20 as a sub- module, and computes the component secret keys, hash keys and the random pads.
    - SWIFFT hash function was used to compute the component public keys and construct the Merkle tree.
  - Signature Generation
    - 1-CFF algorithm to determine the subset of component keys that are associated with a message.
    - The signing also use ChaCha20 and SWIFFT.
  - Signature Verification
    - 1-CFF algorithm to determine the subset of component keys that are associated with a message.
    - The signing and the verification algorithms use ChaCha20, SWIFFT, and the 1-CFF.

Merkle Tree Construction



L_i Tree
(or KSN-OTS Tree)

H-Tree
(or MSS Tree)

K2SN Signature Algorithms

UNIVERSITY OF CALGARY

Key Generation

User inputs index i to get the sk_i from sk secret it already has.

$KeyGenTinyDTLS(sk, i)$ has following input and output:
Input : sk, i
Output : sk_i

Signature Generation

sk_i was generated for signing message i.

Input : sk_i, msg
Output : $sig(i, pk, \mathcal{PK}_i, Auth)$, where :

- i : index of message signed(OTS index)
- pk : Sum of component secret keys(B_{mes})
- $\mathcal{PK}_i$: Set of public component keys for i-th message
- $Auth$: Nodes of the MSS tree for authentication of OTS tree with MSS tree root.

Signature Verification

Signature verification works into two parts :

1. Verify pk against $\mathcal{PK}_i$
2. Verify $\mathcal{PK}_i$ against y_{00}(Root of MSS tree)

SignVerify() has following input and outputs:
Input : msg, sig
Output : True/False

Sabyasachi Karati and Reihaneh Safavi-Naini. K2sn-mss: An efficient post-quantum signature. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 501–514, 2019.

PQ-TESLA

- Threat Model of PQ-TESLA :
  - Adversary having access to a quantum computer.

- *For each TESLA response message, OTS_i is used.*
- *Saves state of the signature- Index of the OTS and Authentication path.*

- In PQ-TESLA, replace DSS with any hash-based signature scheme.
  - Replace ECDSA with K2SN-MSS

- In TESLA Initialization, the Synchronization Message is signed using K2SN-OTS of K2SN-MSS.

- All other sender and receiver operations remains same as described in the background.

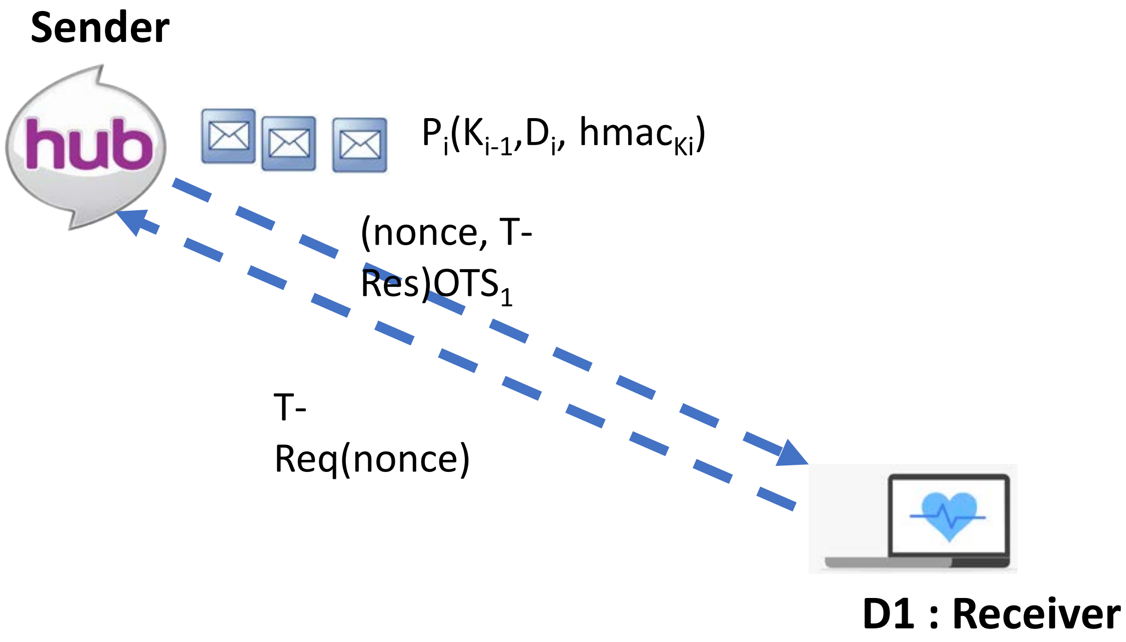
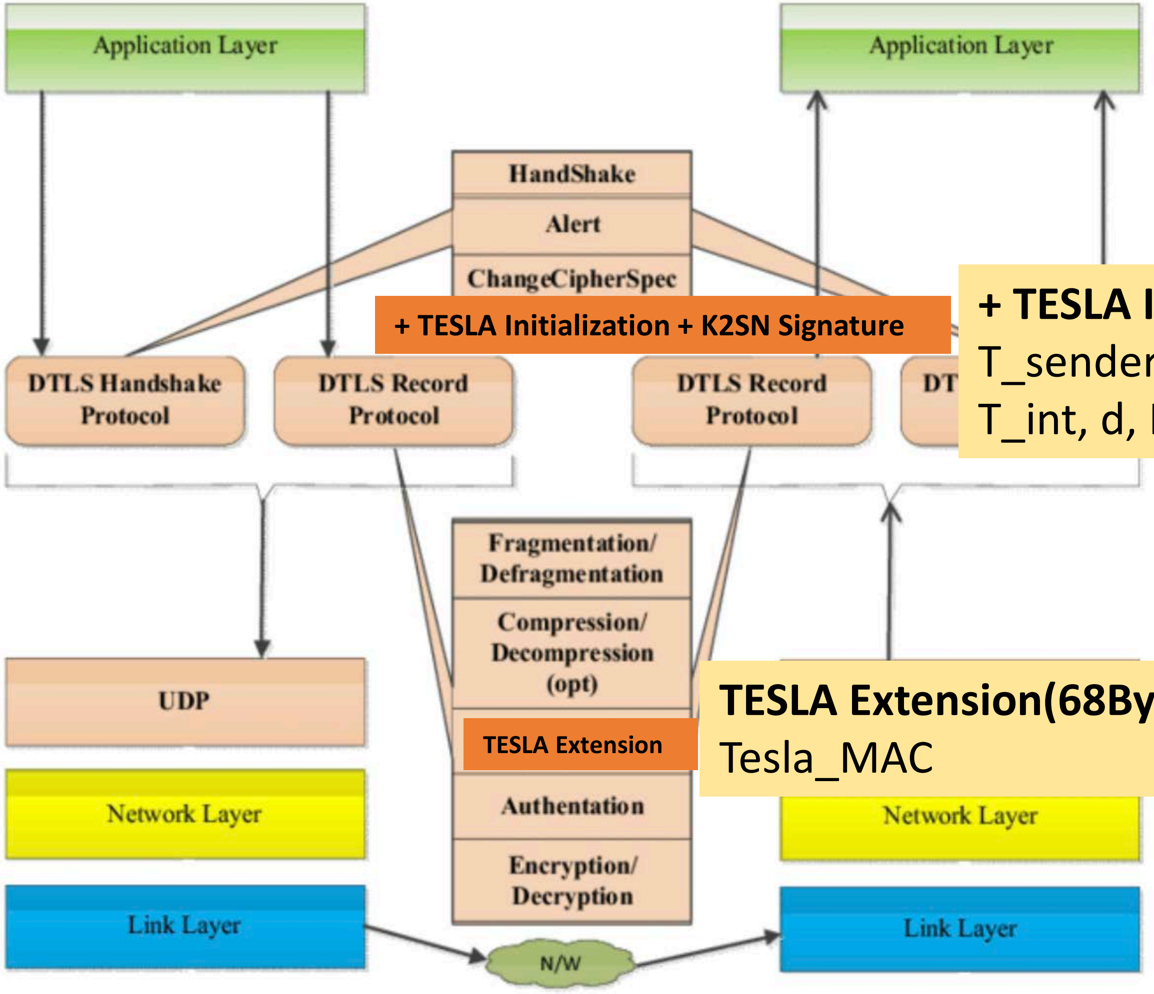


Fig : Using K2SN-MSS in TESLA

PQ TESLA and its Application to DTLS

(High level Overview)



RTT

+ TESLA Initialization(113Bytes) : Nonce, T_sender, rate, interval_id, T_start, n, T_int, d, K_i, ***K2SN Signature(21.3KB)***

TESLA Extension(68Bytes) : i, K_i, Tesla_MAC

ClientKeyExchange + Tesla Synchronization Response (113) + ECDSA(136Bytes)

ServerHello + Tesla Synchronization Request (32 Bytes)

ClientKeyExchange + Tesla Synchronization Response (113 Bytes) + K2SN-MSS(21331Bytes)

PQ-DTLS with Source Authentication and Data Integrity

- Security Goals : We aim to make DTLS PQ secure. We claim that integration of post-quantum TESLA still preserves the security of DTLS.
- To provide DTLS with authentication and integrity with PQ security :
  - Packet Authenticity : Every received packet inherits the sender authentication from the handshake layer, which means that the receiver is ensured that origin of the packet is the same as the one established in the handshake. Use of TESLA to provide source authentication(signing key commitment) at handshake layer using hash-based signature.
  - Packet Integrity : The data in the packet has not been tampered with. Use of TESLA authentication tag to provide integrity at record layer.

- Adversary is/has :
  - Capable of intercepting message eavesdrop, capture, drop, resend, delay, and alter packets.
  - Unlimited storage capabilities, and his computing power is large but not unbounded.
  - Access to a quantum computer capable of running Shor’s quantum algorithm in polynomial time.

Limitation : Nonetheless the adversary cannot invert a pseudorandom function (or distinguish it from a random function) with non-negligible probability.

TESLA Initialization in DTLS Handshake Layer

- The following handshake flights were modified:
  - ServerHello + Tesla Synchronization Request(32 Bytes)
  - ClientKeyExchange + Tesla Synchronization Response(113 Bytes) + :
    - Signed with ECDSA (136 Bytes)
    - Signed with K2SN-MSS (21331 Bytes)

```
/** Structure of the TESLA Synchronization Request. 32 BYTES. */
typedef struct{
    uint8_t nonce[32];
} tesla_request;

/** Structure of the TESLA Synchronization Response. 113 BYTES*/
typedef struct {
    uint8_t nonce[32];          /** 32 BYTES of nonce in request packet */
    uint8_t T_sender[16];       /** 16 BYTES Sender's current time */
    uint8_t rate[4];            /** 4 BYTES Interval rate */
    uint8_t interval_id[4];     /** 4 ByTES : Interval index */
    uint8_t T_start[16];        /** 16  BYTES : Start Time corresponding to beginning of session Unix GMT */
    uint8_t T_int[4];           /** 4 BYTES : interval duration (in seconds) */
    uint8_t dis_delay[1];       /** 1 BYTE: Key Disclosure Delay (in number of intervals)*/
    uint8_t key_chain_len[4];   /** 4 BYTES : Length of key chain */
    uint8_t Key_comm[32];       /** 32 BYTES: Commitment Key */
} tesla_sync;
```

Figure 4.2: TESLA request and response structures

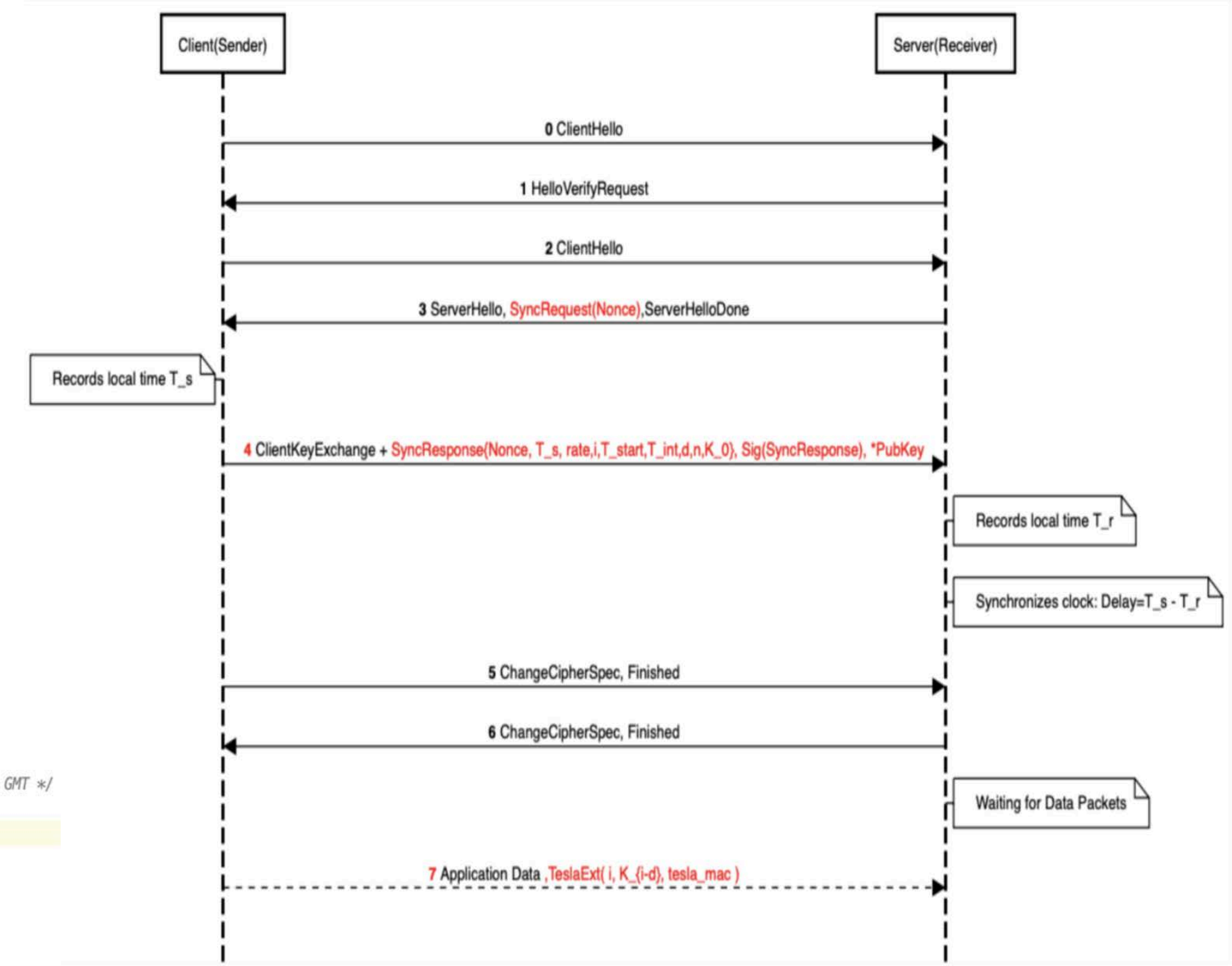


Figure 4.1: Overview of TESLA and its application in TinyDTLS

TESLA Extension in DTLS Record layer

Each application record data has overhead of 68 Bytes added by TESLA extension.
Maximum Payload size: 16384 Or 65536Bytes.

```c
struct packet_store
{
    uint32_t t_id;//[4];
    uint8_t t_msg[10000];
    uint8_t reveal_key[32];
    uint8_t packet_mac[32];
};


typedef struct dtls_peer_t {
  struct dtls_peer_t *next;
  session_t session;          /**< peer address and local interface */
  dtls_peer_type role;        /** DTLS_CLIENT or DTLS_SERVER */
  dtls_state_t state;         /** DTLS engine state */

  dtls_security_parameters_t *security_params[2];
  dtls_handshake_parameters_t *handshake_params;

  uint32_t int_index;            /* Interval-index, increments for every packet */
  uint8_t K[1000][32];           /* 1000 TESLA Key-chain storage */
  uint8_t tesla_mac[32];         /* TESLA HMAC of current packet */
  struct packet_store tesla_ps; /* buffer for storage packet */

} dtls_peer_t;
```

Content type 1 Byte	Version Ma | Mi 2 Byte	Epoch 2 Byte	Seq_number 6 Byte	Length 2 Byte	Ciphertext	MAC

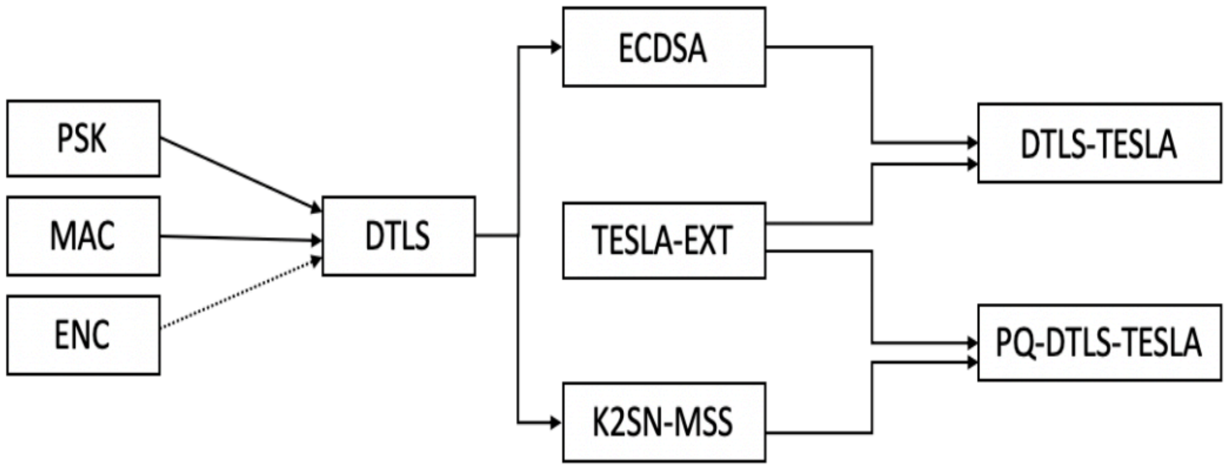
```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               i                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~                         Disclosed Key                         ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
~                           TESLA MAC                           ~
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

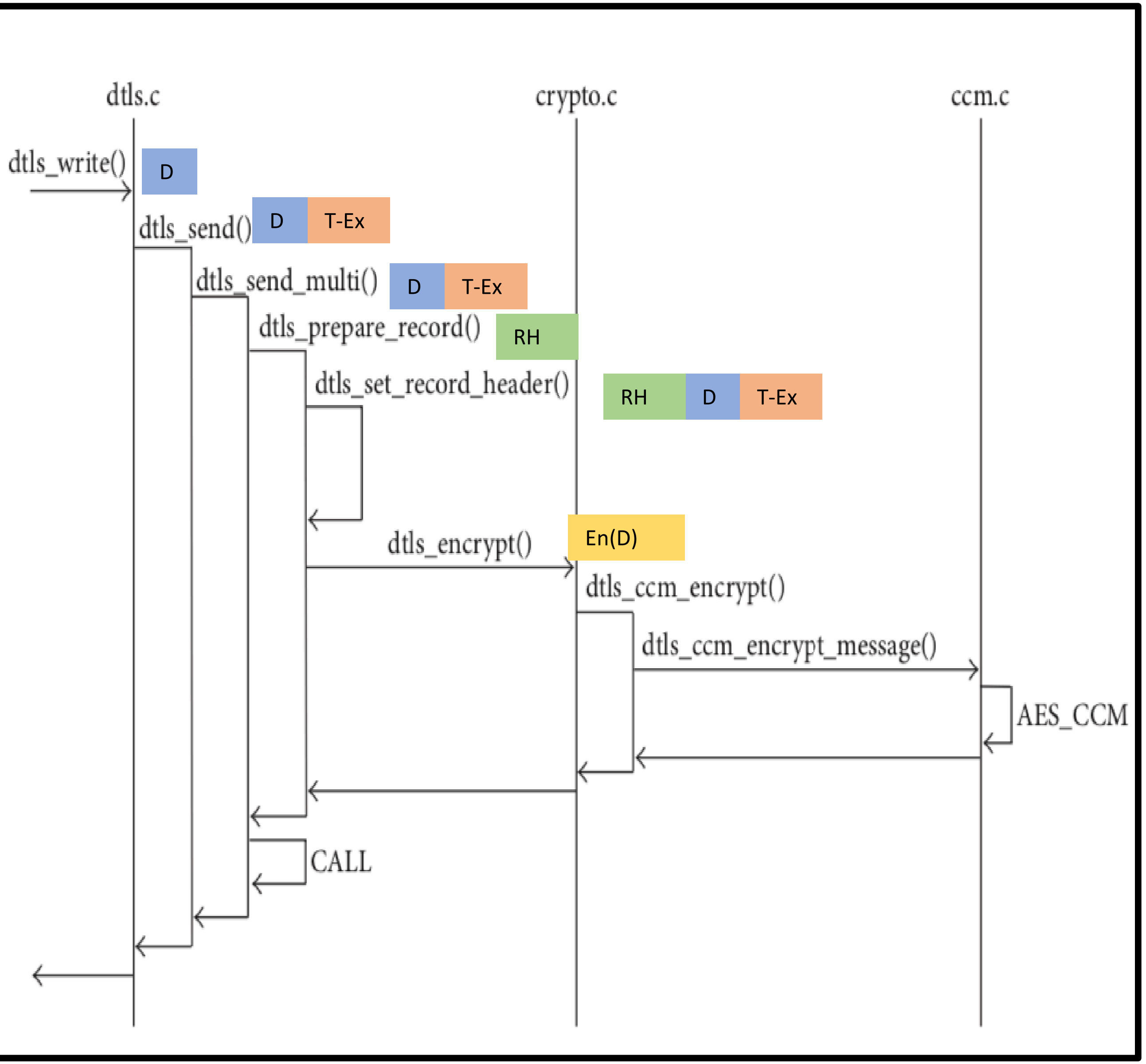
Implementation

TinyDTLS is a light-weight implementation library of the DTLS protocol in C.

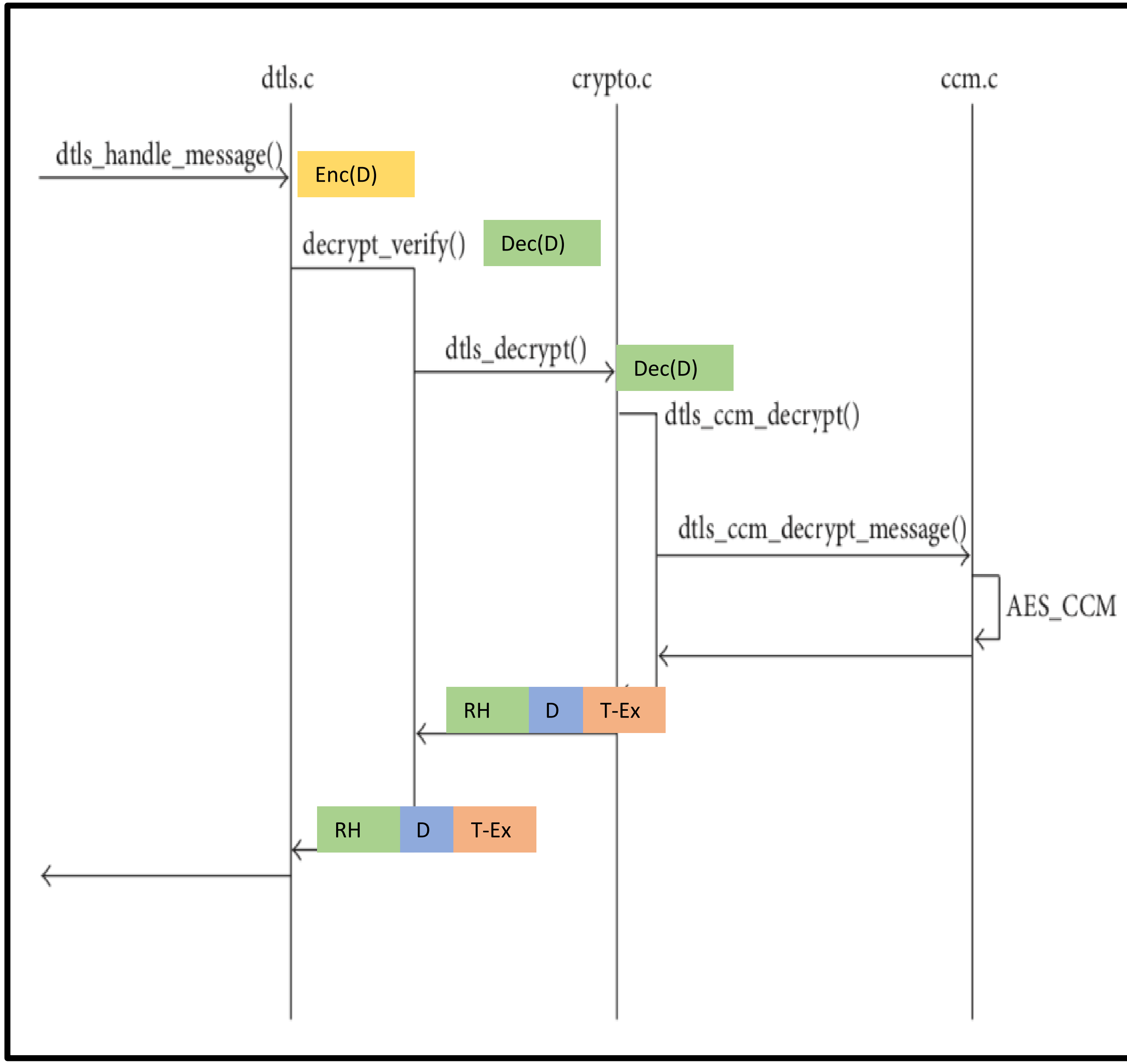
Implemented Protocol

- **DTLS** : PSK, MAC, ENC
- **DTLS-TESLA** : PSK, ENC(optionally), MAC, ECDSA, TESLA-EXT
- **PQ-DTLS-TESLA** : PSK, ENC(optionally), MAC, K2SN-MSS and TESLA-EXT





Sender



Receiver

Function Flow of Sender and Receiver(TinyDTLS) with TESLA

Experimental Setup

Objective:

- How much is overhead of adding PQ security to DTLS ??
- How much time consumed in PQ secure version of DTLS – handshake time and application data transfer time ??
- Computation time for signature schemes –ECDSA and K2SN-MSS.
- Performance Comparison : DTLS vs TESLA to DTLS (without PQ) vs TESLA to DTLS (with PQ).

Testing Environment:

- Client and Server both run on same host computer on Ubuntu 16.04 OS.
- Linux has POSIX support needed to run the TinyDTLS application.
- OS has support AVX2 CPU instructions needed to run K2SN-MSS.

Methodology/Routine:

- Communication is unicast, DTLS Server is in waiting state to accept DTLS client requests.
- Before a DTLS client can initiate the DTLS handshake, it needs to know the IP address of that DTLS server and PSK credentials to use.
- We conduct experiments for 50 DTLS client consecutively sending requests to DTLS server.

We discuss about results of the experiments in three aspects: feasibility, performance, and efficiency.

Experiments

Performance Metric	Experiments	Objective
Feasibility 	Handshake layer overhead	Measure extra bytes to be transferred during HS.
	Record layer overhead	Measure extra bytes to be transferred for each packet.
	Code Size	Measure the is theoretic value of code size measurement from the implemented code in terms of lines of code(loc).
Performance and Efficiency 	Evaluation of cryptographic primitives	We evaluate the performance of ECDSA signature and hash-based signature, K2SN-MSS
	Handshake latency	TESLA initialization and PQ-security to DTLS for the overall duration of a handshake
	Data transfer latency	Compare latency time of authenticated messages.

Experiment 1 : Handshake layer Overhead

- ***Aim.*** *Aim of this experiment are to see the cost of adding post-quantum security to DTLS handshake, in terms of bytes overhead.*

Table 6.1: DTLS Handshake Flights

Flight	**DTLS**	**DTLS-TESLA**	**PQ-DTLS-TESLA**
Client Hello	67	67	67
Hello Verify Request	44	44	44
Client Hello(cookie)	83	83	83
Server Hello	63	95	95
Server Hello Done	25	25	25
Client Key Exchange	42	177	21481
Change Cipher Specs	14	14	14
FINISH(Client)	53	53	53
Change Cipher Specs	14	14	14
FINISH(Server)	53	53	53
Total Bytes	**458**	**625**	**21929**

Table 6.2: TESLA handshake Overhead

Field	**TESLA**	**PQ-TESLA**
Nonce(Request)	32	32
Nonce(Response)	32	32
T_s	16	16
rate	4	4
i	4	4
T_{start}	16	16
T_{int}	4	4
d	1	1
n	4	4
K_0	32	32
Sig	136(ECDSA)	21331(K2SN-MSS)
Total Bytes	**273**	**21476**

Experiment 2 : Record Datagram Overhead

Aim. *Aim of this experiment are to see the cost of adding post-quantum security to record datagram of DTLS*

Table 6.3: Each DTLS packet(or record datagram).

Field	**Bytes**
Content Type	1
Version	2
Epoch	2
Seq Num	6
Length	2
Payload Data	N(variable)
MAC	32
Total	**43+N**

Table 6.4: TESLA per packet overhead.

Field	**Bytes**
Interval Index	4
Disclosed Key	32
TESLA MAC	32
Total	**68**

Experiment 3 : Code Size

***Aim**. The aim was to measure the is theoretic value of code size measurement from the imple-mented code, which is in C programming language*

Table 6.5: Code Size

Module	**Code Size(loc)**[3]
DTLS	11453
TESLA	6824
K2SN-MSS	197467
DTLS-TESLA	18277
PQ-DTLS-TESLA	216183
Application data(Bytes)	Variable

Experiment 4 : Evaluation of cryptographic primitives

Aim : *We evaluate the performance of ECDSA signature and hash-based signature, K2SN on the targeted machines, by measuring the run-time for key generation, signing and verification operations. We want to measure the cost of implementing a hash-based signature in terms of how fast the algorithm takes as compared to a currently used non-post-quantum signature, ECDSA.*

Table 4: Runtime of cryptographic primitives in seconds(Average of 100) in milliseconds

Phase	ECDSA	K2SN
Key Generation	0.00737	497.931
Signature Generation	0.00599	0.001602
Verification	0.0136	0.000013

Experiment 5 : Handshake Latency

***Aim** : We measure the handshake time from the beginning of client hello until the finished message has been received.*

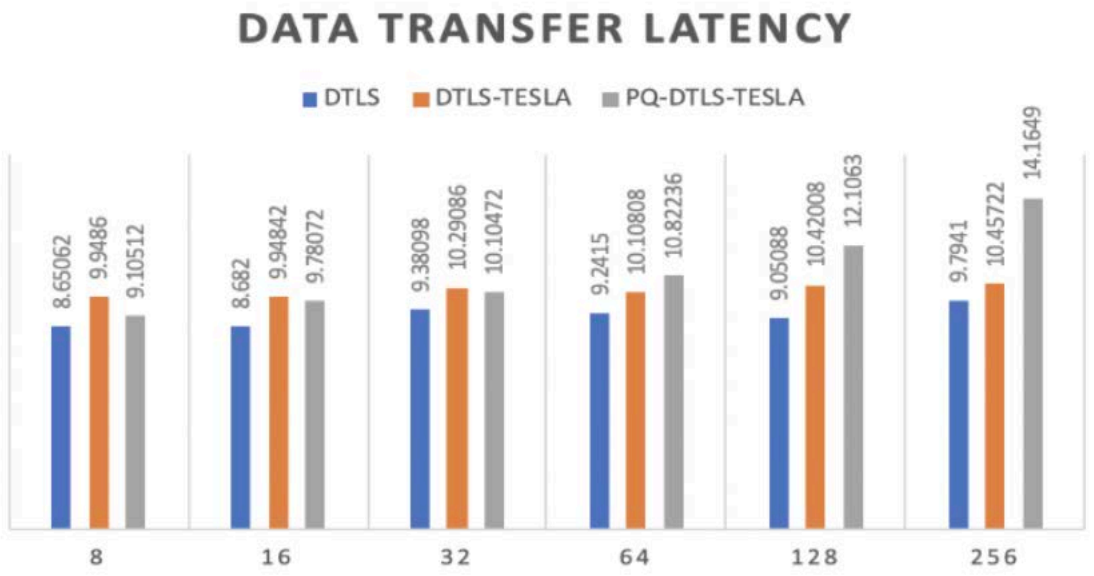
Table 5: Handshake latency: Avg = 50 handshakes versus time in millisecond

Machine Locations	**DTLS**	**DTLS-TESLA**	**PQ-DTLS-TESLA**
UofC Localhost	0.000196	0.035683	497.065846

Experiment 6 : Data Transfer Latency

Aim. *The data latency is considered as the measure of system's cryptographic performance. A packet goes through encryption and integrity at sender's side and decryption and integrity check at receiver's side.*

Payload size	DTLS	DTLS-TESLA	PQ-DTLS-TESLA
8	8.65062	9.9486	9.10512
16	8.682	9.94842	9.78072
32	9.38098	10.29086	10.10472
64	9.2415	10.10808	10.82236
128	9.05088	10.42008	12.1063
256	9.7941	10.45722	14.1649



Experiments & Results

Performance Metric	Experiments	Objective	Results
Feasibility 	Handshake layer overhead	Measure extra bytes to be transferred during HS.	TESLA initialization overhead is 145 Bytes + ECDSA(136Bytes) = 281 Bytes + K2SN-MSS(21331Bytes) = 21476 Bytes
	Record Datagram overhead	Measure extra bytes to be transferred for each packet.	Overhead is 68 Bytes per packet.
	Code Size	Measure the is theoretic value of code size measurement from the implemented code in terms of lines of code(loc).	DTLS : 11453 TESLA : 6824 K2SN-MSS : 197467 DTLS-TESLA : 18277 PQ-DTLS-TESLA : 216183
Performance and Efficiency 	Evaluation of cryptographic primitives	We evaluate the performance of ECDSA signature and hash-based signature, K2SN-MSS	T_{KeyGen} (K2SN-MSS) > T_{KeyGen} (ECDSA) T_{SigGen} (K2SN-MSS) < T_{SigGen} (ECDSA) T_{SigVer}(K2SN-MSS) < T_{SigVer} (ECDSA)
	Handshake latency	TESLA initialization and PQ-security to DTLS for the overall duration of a handshake	T_{HS} (DTLS-TESLA) ~ 64 * T_{HS} (DTLS) T_{HS} (PQ-DTLS-TESLA) ~ 28 * T_{HS} (DTLS)
	Data transfer latency	Compare latency time of authenticated messages.	$T_{processing} \propto$ PayloadSize (Observed) $T_{processing}$ (DTLS-TESLA) > $T_{processing}$ (DTLS) $T_{processing}$ (PQ-DTLS-TESLA) > $T_{processing}$ (DTLS) $T_{processing}$ (DTLS-TESLA) ~ $T_{processing}$ (PQ-DTLS-TESLA)

Conclusion & Future Work

Our integration of quantum-resistant schemes into DTLS proves to be feasible: the induced performance overhead is tolerable, to get PQ compatible protocol.

We provide and analyse the attacks in our modified DTLS that accommodates TESLA and makes DTLS PQ secure, in our next phase, for security and scrutiny of proposed PQ system.

References

[1] A. Perrig, R. Canetti, J. D. Tygar, and D. Song,Efficient authenticationand signing of multicast streams over lossy channels, 2000.

[2] . Perrig, R. Canetti, J. D. Tygar, and D. Song,Efficient authenticationand signing of multicast streams over lossy channels, 2000.

[3] T. Kothmayr, C. Schmitt, W. Hu, M. Br¨unig, and G. Carle,DTLS basedsecurity and two-way authentication for the Internet of Things. Elsevier,2013, vol. 11, no. 8.

[4] S. Karati and R. Safavi-Naini, “K2sn-mss: An efficient post-quantumsignature,” inProceedings of the 2019 ACM Asia Conference onComputer and Communications Security, 2019, pp. 501–514.

[5] H.Technologies.(2019) Whitepaper: Partnering with industry for 5g security assurance.

[Online] https://www-file.huawei.com/-/media/corporate/pdf/trust-center/huawei-5g-security-white-paper-4th.pdf

Comments & Suggestions

- Key Storage for TESLA – how much storage is required?
- Why we use F and F’
- Emphasize on HS latency
- How to calculate upper bound on sender’s interval,(value x??)
- In the PQ DTLS, did you use certificate? If not, what did you do to replace the certificate?
- RAM used before and after compiling