

# QuxTech PQC VoIP Security Module - Installation & Usage Manual

## Post-Quantum VoIP Security Module

### Installation & Usage Manual

**Package:** @quxtech/pqc-crypto **Version:** 1.0.0 **Module:** voip

### Table of Contents

- 1. [Overview](#)
- 2. [Installation](#)
- 3. [Quick Start](#)
- 4. [API Reference](#)
- 5. [Call Flow](#)
- 6. [Security Architecture](#)
- 7. [Integration Examples](#)
- 8. [Troubleshooting](#)

### Overview

The VoIP module provides post-quantum secure voice communications using:

Component	Algorithm	Purpose
Key Exchange	ML-KEM (FIPS 203)	Quantum-resistant key encapsulation
Authentication	ML-DSA (FIPS 204)	Digital signatures for call verification
Voice Encryption	AES-256-GCM	Authenticated encryption for voice frames
Key Derivation	HKDF-SHA3	SRTP-compatible key derivation

### Security Levels

Level	KEM	DSA	AES Equivalent
-------	-----	-----	----------------

3	ML-KEM-768	ML-DSA-65	AES-192
5	ML-KEM-1024	ML-DSA-87	AES-256

---

## Installation

### npm (Recommended)

```
npm install @quxtech/pqc-crypto
```

### Yarn

```
yarn add @quxtech/pqc-crypto
```

### pnpm

```
pnpm add @quxtech/pqc-crypto
```

## Requirements

- Node.js 18.x or 20.x+
  - TypeScript 5.0+ (for TypeScript projects)
- 

## Quick Start

### Basic Secure Call Setup

```
import { voip } from '@quxtech/pqc-crypto';

// === CALLER SIDE ===

// 1. Generate caller's key pair
const callerKeys = voip.generateKeyPair('5');

// 2. Create call request
const request = voip.createCallRequest(callerKeys, 'OPUS', {
  callerName: 'Alice',
  callerNumber: '+1234567890'
});

// Send `request` to callee via signaling channel...

// === CALLEE SIDE ===

// 3. Generate callee's key pair
const calleeKeys = voip.generateKeyPair('5');

// 4. Verify and accept the call
if (voip.verifyCallRequest(request, '5')) {
  const { response, session: calleeSession } =
voip.acceptCall(request, calleeKeys);
  // Send `response` back to caller...
}
```

```

// === CALLER SIDE (continued) ===

// 5. Complete call establishment
const callerSession = voip.completeCall(request, response,
callerKeys);

// Both parties now have matching encryption keys!
console.log('Call established:', callerSession.callId);

// === VOICE TRANSMISSION ===

// Encrypt a voice frame (sender)
const voiceData = new Uint8Array([/* PCM/Opus encoded audio */]);
const encryptedFrame = voip.encryptFrame(
  callerSession.callId,
  voiceData,
  Date.now()
);

// Decrypt a voice frame (receiver)
const decryptedFrame = voip.decryptFrame(
  calleeSession.callId,
  encryptedFrame
);

// Access the original audio
const audioPayload = decryptedFrame.payload;

```

---

## API Reference

### Key Generation

#### **generateKeyPair(securityLevel?)**

Generate a VoIP key pair containing KEM and DSA keys.

```
function generateKeyPair(securityLevel?: '3' | '5'): VoIPKeyPair;
```

**Parameters:** - securityLevel - NIST security level (default: '5')

**Returns:** VoIPKeyPair

```
interface VoIPKeyPair {
  kem: { publicKey: string; secretKey: string };
  dsa: { publicKey: string; secretKey: string };
  securityLevel: '3' | '5';
}
```

#### **Example:**

```

const keys = voip.generateKeyPair('5');
console.log('KEM Public Key length:', keys.kem.publicKey.length);
// ML-KEM-1024: 3168 hex chars (1584 bytes)

```

---

### **generateCallId()**

Generate a unique 128-bit call identifier.

```
function generateCallId(): string;
```

**Returns:** 32-character hex string

---

### **generateSSRC()**

Generate a random SSRC (Synchronization Source) for RTP.

```
function generateSSRC(): number;
```

**Returns:** 32-bit unsigned integer

---

## **Call Establishment**

### **createCallRequest(callerKeys, codec?, metadata?)**

Create a signed call request (equivalent to SIP INVITE).

```
function createCallRequest(  
  callerKeys: VoIPKeyPair,  
  codec?: VoIPCodec,  
  metadata?: Record<string, unknown>  
) : VoIPCallRequest;
```

**Parameters:** - callerKeys - Caller's VoIP key pair - codec - Audio codec: 'OPUS', 'G711', 'G722', 'G729', 'PCMU', 'PCMA' - metadata - Optional call metadata (caller name, SDP, etc.)

**Returns:** VoIPCallRequest

```
interface VoIPCallRequest {  
  callId: string;  
  callerKemPublicKey: string;  
  callerDsaPublicKey: string;  
  timestamp: number;  
  signature: string;  
  codec?: VoIPCodec;  
  metadata?: Record<string, unknown>;  
}
```

---

### **verifyCallRequest(request, securityLevel?)**

Verify the digital signature on a call request.

```
function verifyCallRequest(  
  request: VoIPCallRequest,  
  securityLevel?: '3' | '5'  
) : boolean;
```

**Returns:** true if signature is valid

---

### **acceptCall(request, calleeKeys)**

Accept an incoming call and establish the session.

```
function acceptCall(  
  request: VoIPCallRequest,  
  calleeKeys: VoIPKeyPair  
) : { response: VoIPCallResponse; session: VoIPSession };
```

**Throws:** Error if request signature is invalid

**Returns:** - response - Signed response to send back to caller - session - Established session with encryption keys

---

**completeCall(request, response, callerKeys)**

Complete call establishment on the caller side.

```
function completeCall(  
  request: VoIPCallRequest,  
  response: VoIPCallResponse,  
  callerKeys: VoIPKeyPair  
) : VoIPSession;
```

**Throws:** Error if response signature is invalid

---

## Frame Encryption

**encryptFrame(callId, payload, timestamp)**

Encrypt a voice frame for transmission.

```
function encryptFrame(  
  callId: string,  
  payload: Uint8Array,  
  timestamp: number  
) : VoIPEncryptedFrame;
```

**Parameters:** - callId - Active call identifier - payload - Raw audio data (PCM, Opus, etc.) - timestamp - RTP timestamp

**Returns:** VoIPEncryptedFrame

```
interface VoIPEncryptedFrame {  
  sequenceNumber: number;  
  timestamp: number;  
  ssrc: number;  
  nonce: string;  
  ciphertext: string;  
  authTag: string;  
}
```

**Throws:** - Error if no active session - Error if session is on hold or terminated

---

**decryptFrame(callId, frame)**

Decrypt a received voice frame.

```
function decryptFrame(  
    callId: string,  
    frame: VoIPEncryptedFrame  
): VoIPDecryptedFrame;
```

**Returns:** VoIPDecryptedFrame

```
interface VoIPDecryptedFrame {  
    sequenceNumber: number;  
    timestamp: number;  
    ssrc: number;  
    payload: Uint8Array;  
}
```

**Throws:** - Error if authentication fails (tampered data) - Error if no active session

---

## Session Management

### getSession(callId)

Retrieve an active session.

```
function getSession(callId: string): VoIPSession | null;
```

---

### holdCall(callId)

Put a call on hold. Frames cannot be encrypted/decrypted while on hold.

```
function holdCall(callId: string): void;
```

---

### resumeCall(callId)

Resume a call from hold.

```
function resumeCall(callId: string): void;
```

---

### terminateCall(callId)

End a call and clean up resources.

```
function terminateCall(callId: string): VoIPSessionStats | null;
```

**Returns:** Final call statistics or null if session not found

---

### getActiveCallCount()

Get number of active (connected or on hold) calls.

```
function getActiveCallCount(): number;
```

---

### getActiveCallIds()

Get list of all active call IDs.

```
function getActiveCallIds(): string[];
```

---

## Statistics

**getSessionStats(callId)**

Get current session statistics.

```
function getSessionStats(callId: string): VoIPSessionStats | null;
```

**Returns:** VoIPSessionStats

```
interface VoIPSessionStats {  
  callId: string;  
  duration: number;           // milliseconds  
  framesSent: number;  
  framesReceived: number;  
  bytesEncrypted: number;  
  bytesDecrypted: number;  
  packetsLost: number;  
}
```

---

**reportPacketLoss(callId, count?)**

Report detected packet loss for statistics tracking.

```
function reportPacketLoss(callId: string, count?: number): void;
```

---

## Utilities

**estimateBandwidth(codec?, frameSize?, sampleRate?)**

Estimate bandwidth usage including encryption overhead.

```
function estimateBandwidth(  
  codec?: VoIPCodec,  
  frameSize?: number,  
  sampleRate?: number  
): number;
```

**Returns:** Estimated bytes per second

**Example:**

```
const bandwidth = voip.estimateBandwidth('OPUS', 960, 48000);  
console.log(`Estimated bandwidth: ${bandwidth} bytes/sec`);  
// ~6000-7000 bytes/sec for OPUS
```

---

**getAlgorithmInfo(securityLevel?)**

Get cryptographic algorithm details.

```
function getAlgorithmInfo(securityLevel?: '3' | '5'): {  
  kem: string;
```

```

    dsa: string;
    symmetric: string;
    keyExchangeSize: number;
    signatureSize: number;
};

```

### Example:

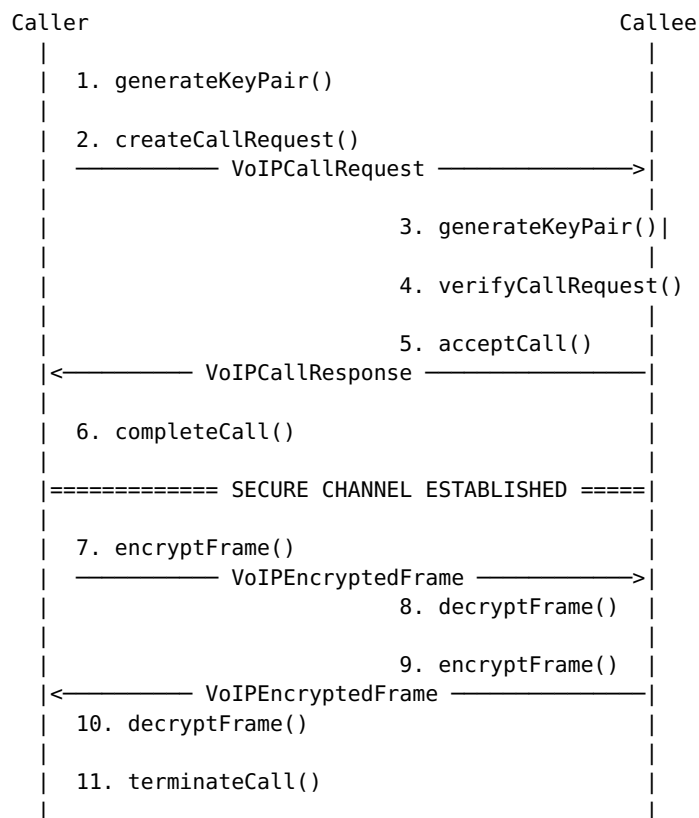
```

const info = voip.getAlgorithmInfo('5');
// {
//   kem: 'ML-KEM-1024',
//   dsa: 'ML-DSA-87',
//   symmetric: 'AES-256-GCM',
//   keyExchangeSize: 1568,
//   signatureSize: 4627
// }

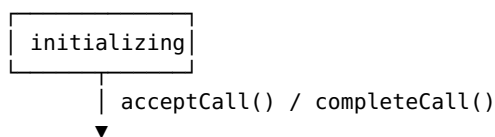
```

## Call Flow

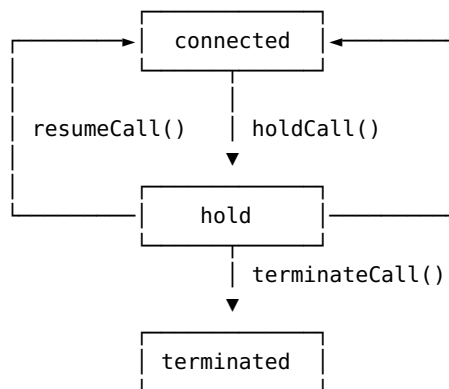
### Sequence Diagram



### State Machine

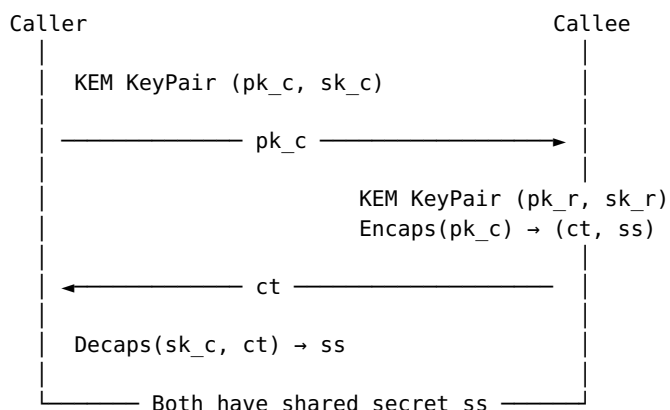




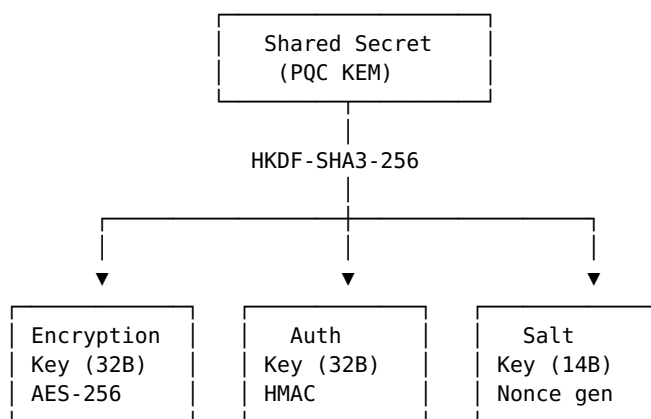


## Security Architecture

### Key Exchange (ML-KEM)



### SRTP Key Derivation



### Frame Encryption

Each voice frame is encrypted using AES-256-GCM with:

- **Nonce Construction:** Salt XOR (SSRC || Packet Index)

- **Packet Index:**  $(\text{ROC} \times 65536) + \text{Sequence Number}$
- **Authentication Tag:** 128-bit GCM tag

Encrypted Frame				
Seq (16b)	TS (32b)	SSRC (32b)	Ciphertext (variable)	Auth Tag (128b)

## Integration Examples

### WebRTC Integration

```
import { voip } from '@quxtech/pqc-crypto';

class PQCWebRTCAdapter {
  private keys: ReturnType<typeof voip.generateKeyPair>;
  private callId: string | null = null;

  constructor(securityLevel: '3' | '5' = '5') {
    this.keys = voip.generateKeyPair(securityLevel);
  }

  // Integrate with WebRTC data channel for signaling
  async initiateCall(dataChannel: RTCDataChannel): Promise<void> {
    const request = voip.createCallRequest(this.keys, 'OPUS');

    dataChannel.send(JSON.stringify({
      type: 'pqc-call-request',
      payload: request
    }));

    this.callId = request.callId;
  }

  // Handle incoming signaling message
  handleSignaling(message: any): void {
    if (message.type === 'pqc-call-request') {
      const { response, session } = voip.acceptCall(message.payload,
this.keys);
      this.callId = session.callId;
      // Send response back...
    } else if (message.type === 'pqc-call-response') {
      voip.completeCall(/* ... */);
    }
  }

  // Transform outgoing audio
  encryptAudio(pcmData: Uint8Array, timestamp: number):
VoIPEncryptedFrame {
    if (!this.callId) throw new Error('No active call');
    return voip.encryptFrame(this.callId, pcmData, timestamp);
  }

  // Transform incoming audio
  decryptAudio(frame: VoIPEncryptedFrame): Uint8Array {
    if (!this.callId) throw new Error('No active call');
  }
}
```

```

        return voip.decryptFrame(this.callId, frame).payload;
    }
}

```

## SIP Integration

```

import { voip } from '@quxtech/pqc-crypto';

// Embed PQC data in SIP headers
function createSIPInvite(
  callerKeys: ReturnType<typeof voip.generateKeyPair>,
  sipInvite: string
): { sipInvite: string; pqcRequest: VoIPCallRequest } {
  const pqcRequest = voip.createCallRequest(callerKeys, 'OPUS');

  // Add custom SIP header with PQC data
  const pqcHeader = `X-PQC-Request:
${Buffer.from(JSON.stringify(pqcRequest)).toString('base64')}`;

  const modifiedInvite = sipInvite.replace(
    '\r\n\r\n',
    `\r\n${pqcHeader}\r\n\r\n`
  );

  return { sipInvite: modifiedInvite, pqcRequest };
}

// Extract and verify PQC data from SIP 200 OK
function processSIP200OK(
  sipResponse: string,
  originalRequest: VoIPCallRequest,
  callerKeys: ReturnType<typeof voip.generateKeyPair>
): VoIPSession {
  const match = sipResponse.match(/X-PQC-Response: (.+)\r\n/);
  if (!match) throw new Error('No PQC response in SIP 200 OK');

  const pqcResponse = JSON.parse(Buffer.from(match[1],
'base64').toString());
  return voip.completeCall(originalRequest, pqcResponse,
callerKeys);
}

```

## Multi-Call Management

```

import { voip } from '@quxtech/pqc-crypto';

class CallManager {
  private keys: ReturnType<typeof voip.generateKeyPair>;

  constructor() {
    this.keys = voip.generateKeyPair('5');
  }

  // List all active calls
  listCalls(): Array<{ callId: string; state: string; duration:
number }> {
    return voip.getActiveCallIds().map(callId => {
      const session = voip.getSession(callId);
      const stats = voip.getSessionStats(callId);
    });
  }
}

```

```

        return {
            callId,
            state: session?.state ?? 'unknown',
            duration: stats?.duration ?? 0
        };
    });
}

// Conference call: mix multiple call audio
mixAudio(frames: Map<string, VoIPEncryptedFrame>): Uint8Array {
    const pcmBuffers: Uint8Array[] = [];

    for (const [callId, frame] of frames) {
        const decrypted = voip.decryptFrame(callId, frame);
        pcmBuffers.push(decrypted.payload);
    }

    // Mix PCM audio (simplified)
    return this.mixPCM(pcmBuffers);
}

private mixPCM(buffers: Uint8Array[]): Uint8Array {
    // Audio mixing implementation...
    return new Uint8Array(/* mixed audio */);
}
}

```

---

## Troubleshooting

### Common Errors

Error	Cause	Solution
Invalid call request signature	Tampered request or wrong security level	Verify both parties use same security level
Invalid call response signature	Tampered response or mismatched keys	Check key pairs are consistent
No active session for call	Session terminated or never established	Verify call establishment completed
Session not in connected state	Trying to encrypt/decrypt while on hold	Call resumeCall() first
Frame decryption failed - authentication error	Tampered frame or wrong keys	Check frame integrity, verify session

### Debugging

```

// Enable detailed logging
const session = voip.getSession(callId);
console.log('Session state:', session?.state);
console.log('Sequence number:', session?.sequenceNumber);

```

```
console.log('Rollover counter:', session?.rolloverCounter);

// Check statistics
const stats = voip.getSessionStats(callId);
console.log('Frames sent:', stats?.framesSent);
console.log('Frames received:', stats?.framesReceived);
console.log('Packets lost:', stats?.packetsLost);

// Algorithm info
const algo = voip.getAlgorithmInfo('5');
console.log('Using algorithms:', algo);
```

Performance Considerations

- 1. **Key Generation:** ML-KEM-1024 and ML-DSA-87 key generation takes ~10-50ms. Generate keys during app initialization, not during call setup.
- 2. **Frame Encryption:** AES-256-GCM encryption is fast (~1µs per frame). The VoIP module is suitable for real-time audio.
- 3. **Memory:** Each active session uses ~5KB. The session store is in-memory; for many concurrent calls, monitor memory usage.
- 4. **Sequence Rollover:** The module handles 16-bit sequence number rollover automatically via the ROC (Rollover Counter).

Codec Reference

Codec	Bitrate	Frame Size	Use Case
OPUS	6-510 kbps	2.5-60ms	Recommended for most uses
G.711 (PCMU/PCMA)	64 kbps	20ms	Legacy PSTN interop
G.722	64 kbps	20ms	Wideband audio
G.729	8 kbps	10ms	Low bandwidth

Version History

Version	Changes
1.0.0	Initial release with ML-KEM/ML-DSA

License

MIT OR Apache-2.0