Interledger Protocol (ILP) Cryptography Labs

Complete Lab Guide with Python Code – Prepared by Dr. Stanley Githinji

Introduction

The Interceder Protocol (ILP) enables secure payments across different ledgers and networks using cryptographic primitives. At its core, ILP uses **crypto-conditions** - a set of cryptographic puzzles that lock and unlock payments.

Overview of Cryptography

Symmetric Key Cryptography (SKC) - is primarily used in the STREAM protocol, which assumes parties start with a shared symmetric key to authenticate any number of subsequent packets. This shared secret enables efficient generation of conditions (hashes) and fulfillments (HMACs) for multiple payments without repeated asymmetric operations . The Shared Secret Protocol allows sender and receiver to remain stateless while generating conditions of the form `SHA256(HMAC_SHA256(shared_secret, payment_info))` .

Public Key Cryptography (PKC) serves two main functions in ILP. First, it enables secure key exchange to establish the shared symmetric secrets used by STREAM—typically accomplished through TLS or similar mechanisms . Second, PKC is formalized in crypto-conditions, which define logical circuits where fulfillments can be validated by verifying cryptographic signatures. ILP supports RSA-SHA-256 and ED25519-SHA256 as standard condition types . PKC also enables passive transport modes where senders encrypt payment details using recipient public keys .

Hash Functions (specifically SHA-256) form the bedrock of ILP's conditional transfer mechanism. They are used in Hashed-Timelock Agreements (HTLAs) , the generalization of HTLCs that secure multi-hop payments . The workflow is:

- A hashlock `H = SHA256(preimage P)` is created, with preimage `P` known only to the ultimate recipient

- Funds are conditionally transferred along the payment path requiring presentation of `P`

- All participants use the same hashlock, ensuring atomicity—either all transfers succeed or none do

The PREIMAGE-SHA-256crypto-condition type directly implements this mechanism, where presenting the valid preimage fulfills the condition . **This cryptographic escrow approach eliminates intermediary risk connectors cannot lose or steal funds as they only unlock payments upon preimage presentation**

Why Cryptography Matters in ILP

- Atomicity: Ensures transactions either complete fully or not at all
- Security: Protects funds during multi-hop transfers
- Interoperability: Allows different ledgers to communicate securely
- Trust Minimization: Removes need for central authorities

🔑 **Core Cryptographic Concepts**

Hash Functions (SHA-256)- Produces 256-bit hashes that are deterministic and one-way | Creating conditions from secrets

Preimage- The secret value before hashing | Fulfilling hashlock conditions |

Conditions - Cryptographic puzzles that lock payments | Shared publicly to lock funds |

Fulfillments- Solutions to cryptographic puzzles | Used to unlock and claim payments |

Public Key -Shareable key for verification | Identifying wallet addresses |

Private Key - Secret key for signing | Authorizing payments

🎁 Required Libraries

```bash
# Core cryptographic libraries
pip install pynacl              # For Ed25519 signatures
pip install cryptography        # For encryption and key management
pip install base58              # For base58 encoding
pip install python-dotenv       # For loading .env files
```

`🔬 Lab 1: Hashlocks and Preimage Conditions

Theory

Hashlocks are the simplest form of crypto-conditions. A sender creates a condition by hashing a random secret (preimage). The payment is released only when someone provides the original secret.

```
Preimage (Secret) → SHA-256 → Condition (Hash)
Condition + Preimage → Verification (True/False)
```

Complete Code

```python
"""
```

Lab 1: Hashlocks and Preimage Conditions

Learn how ILP uses hash-based conditions to lock payments

"""

```
import hashlib
import os
import base64
import time


class HashlockPayment:
    """
    Simulates an ILP hashlock payment system
    """

    def __init__(self):
        self.secrets = {}  # Store secrets temporarily (in production, don't store!)
        self.conditions = {}

    def create_hashlock(self):
        """
        Create a hashlock condition and secret

        Returns:
            dict: Contains payment_id, condition (public), and secret (private)
        """
        # Generate random 32-byte secret (256 bits)
        secret = os.urandom(32)
        secret_b64 = base64.urlsafe_b64encode(secret).decode('utf-8')

        # Create condition (hash of secret)
```

```
    condition = hashlib.sha256(secret).digest()
    condition_b64 = base64.urlsafe_b64encode(condition).decode('utf-8')

    # Store for verification
    payment_id = f"payment_{int(time.time())}"
    self.secrets[payment_id] = secret_b64
    self.conditions[payment_id] = condition_b64

    print(f"\n🔐 Payment ID: {payment_id}")
    print(f"📌 Condition (public): {condition_b64}")
    print(f"🔑 Secret (private - DO NOT SHARE): {secret_b64}")

    return {
        'payment_id': payment_id,
        'condition': condition_b64,
        'secret': secret_b64
    }

def fulfill_hashlock(self, payment_id, provided_secret):
    """
    Attempt to fulfill a hashlock by providing the secret

    Args:
        payment_id: ID of the payment
        provided_secret: Secret to test

    Returns:
        dict: Result of fulfillment attempt
    """
    if payment_id not in self.conditions:
```

```
        return {'success': False, 'error': 'Payment not found'}

    stored_condition = self.conditions[payment_id]

    # Recreate condition from provided secret
    try:
        secret_bytes = base64.urlsafe_b64decode(provided_secret)
        calculated_condition = hashlib.sha256(secret_bytes).digest()
        calculated_b64 = base64.urlsafe_b64encode(calculated_condition).decode('utf-8')
    except Exception as e:
        return {'success': False, 'error': f'Invalid secret format: {e}'}

    # Verify
    if calculated_b64 == stored_condition:
        print(f"\n✓ Payment {payment_id} fulfilled successfully!")
        return {'success': True, 'message': 'Payment released'}
    else:
        print(f"\n✗ Invalid secret for payment {payment_id}")
        return {'success': False, 'error': 'Invalid secret'}

def simulate_payment_flow(self):
    """
    Simulate a complete payment flow with sender and receiver
    """
    print("\n" + "="*60)
    print("SIMULATING ILP HASHLOCK PAYMENT")
    print("="*60)

    # Step 1: Receiver creates a condition
    print("\n📤 Step 1: Receiver creates condition")
```

```
        payment = self.create_hashlock()

        # Step 2: Receiver sends condition to sender
        print("\n📨 Step 2: Receiver sends condition to sender")
        print(f"   Condition shared: {payment['condition'][:20]}...")

        # Step 3: Sender locks funds with condition
        print("\n💰 Step 3: Sender locks funds using condition")
        print(f"   Funds locked with condition: {payment['condition'][:20]}...")

        # Step 4: Receiver fulfills by revealing secret
        print("\n🔓 Step 4: Receiver fulfills with secret")
        result = self.fulfill_hashlock(payment['payment_id'], payment['secret'])

        return result

def run_lab1():
    """Main function to run Lab 1 exercises"""
    print("\n" + "="*60)
    print("LAB 1: HASHLOCKS AND PREIMAGE CONDITIONS")
    print("="*60)

    payment_system = HashlockPayment()

    # Exercise 1: Basic Hashlock
    print("\n📝 EXERCISE 1: Create Your Own Hashlock")
    print("-"*40)

    my_payment = payment_system.create_hashlock()
```

```
    # Try to fulfill with correct secret
    print("\nTrying correct secret...")
    payment_system.fulfill_hashlock(my_payment['payment_id'], my_payment['secret'])

    # Try to fulfill with wrong secret
    wrong_secret = base64.urlsafe_b64encode(b"wrong_secret").decode('utf-8')
    print("\nTrying wrong secret...")
    payment_system.fulfill_hashlock(my_payment['payment_id'], wrong_secret)

    # Exercise 2: Complete Flow
    payment_system.simulate_payment_flow()

    # Exercise 3: Explore Properties
    print("\n📝 EXERCISE 3: Hash Properties")
    print("-"*40)

    test_strings = ["same", "same", "different"]
    for s in test_strings:
        hash_val = hashlib.sha256(s.encode()).digest()
        print(f"'{s}' -> {base64.b64encode(hash_val)[:20]}...")

    print("\n✨ Notice: Same input = Same hash, Different input = Different hash")

if __name__ == "__main__":
    run_lab1()
```

Lab 1 Questions

1. What happens if you change one byte of the secret?

2. Why is it safe to share the condition but not the secret?

3. How would you prevent replay attacks using the same secret twice?

🔏 Lab 2: Ed25519 Signatures and Key Management

Theory

ILP wallets use Ed25519 keypairs. The **private key** signs transactions, while the **public key** verifies signatures.

```

Private Key → Sign(Transaction) → Signature

Public Key + Transaction + Signature → Verify(Valid/Invalid)

```

Complete Code

```python
"""
Lab 2: Ed25519 Signatures and Key Management
Learn how ILP uses digital signatures for authentication
"""

import nacl.signing
import nacl.encoding
import hashlib
import base64
import json
import time

class Ed25519Wallet:
    """
    Simulates an ILP wallet using Ed25519 signatures
    """
```

```
def __init__(self, wallet_name="default"):
    self.wallet_name = wallet_name

    # Generate new keypair using libsodium/nacl
    self.signing_key = nacl.signing.SigningKey.generate()
    self.verify_key = self.signing_key.verify_key

    # Store in different formats
    self.private_key_hex = self.signing_key.encode(
        encoder=nacl.encoding.HexEncoder
    ).decode('utf-8')

    self.public_key_hex = self.verify_key.encode(
        encoder=nacl.encoding.HexEncoder
    ).decode('utf-8')

    self.public_key_b64 = base64.urlsafe_b64encode(
        self.verify_key.encode()
    ).decode('utf-8')

    print(f"\n🔑 Created wallet: {wallet_name}")
    print(f"📌 Public Key (hex): {self.public_key_hex[:32]}...")
    print(f"📌 Public Key (b64): {self.public_key_b64[:20]}...")
    print(f"🔐 Private Key (hex): {self.private_key_hex[:32]}... (KEEP SECRET!)")

def sign_transaction(self, transaction_data):
    """
    Sign an ILP transaction with private key
```

```
Args:
    transaction_data: dict or str containing transaction details

Returns:
    dict: Signed transaction with signature and condition
"""
# Convert transaction to bytes
if isinstance(transaction_data, dict):
    tx_bytes = json.dumps(transaction_data, sort_keys=True).encode()
else:
    tx_bytes = transaction_data.encode()

# Create signature
signed = self.signing_key.sign(tx_bytes)
signature = signed.signature
signature_b64 = base64.urlsafe_b64encode(signature).decode('utf-8')

# Create condition (hash of public key + transaction)
condition_data = self.public_key_hex + tx_bytes.hex()
condition = hashlib.sha256(condition_data.encode()).digest()
condition_b64 = base64.urlsafe_b64encode(condition).decode('utf-8')

return {
    'transaction': transaction_data,
    'signature': signature_b64,
    'condition': condition_b64,
    'public_key': self.public_key_hex
}
```

```
def verify_signature(self, transaction, signature_b64, public_key_hex):
    """
    Verify a transaction signature

    Args:
        transaction: Original transaction data
        signature_b64: Signature to verify
        public_key_hex: Public key of signer

    Returns:
        bool: True if signature is valid
    """
    try:
        # Reconstruct verify key
        verify_key_bytes = bytes.fromhex(public_key_hex)
        verify_key = nacl.signing.VerifyKey(verify_key_bytes)

        # Decode signature
        signature = base64.urlsafe_b64decode(signature_b64)

        # Prepare transaction bytes
        if isinstance(transaction, dict):
            tx_bytes = json.dumps(transaction, sort_keys=True).encode()
        else:
            tx_bytes = transaction.encode()

        # Verify
        verify_key.verify(tx_bytes, signature)

        print(f"\n✓ Signature VALID")
```

```
            return True

        except Exception as e:
            print(f"\n✗ Signature INVALID: {e}")
            return False

    def create_payment_condition(self, destination_pubkey, amount):
        """
        Create an ILP payment condition

        Args:
            destination_pubkey: Recipient's public key
            amount: Payment amount

        Returns:
            dict: Payment condition and details
        """
        payment_details = {
            'destination': destination_pubkey[:16],  # Truncate for display
            'amount': amount,
            'timestamp': str(int(time.time())),
            'wallet': self.public_key_hex[:16]  # Truncate for display
        }

        # Sign the payment details
        signed = self.sign_transaction(payment_details)

        return {
            'condition': signed['condition'],
            'details': payment_details,
```

```
            'signature': signed['signature']
        }

def run_lab2():
    """Main function to run Lab 2 exercises"""
    print("\n" + "="*60)
    print("LAB 2: ED25519 SIGNATURES AND KEY MANAGEMENT")
    print("="*60)

    # Exercise 1: Create and Examine Keys
    print("\n📝 EXERCISE 1: Create Your First Wallet")
    print("-"*40)

    wallet1 = Ed25519Wallet("Alice")
    wallet2 = Ed25519Wallet("Bob")

    print(f"\n🔍 Key lengths:")
    print(f"Private key: {len(wallet1.private_key_hex)} hex chars (32 bytes)")
    print(f"Public key: {len(wallet1.public_key_hex)} hex chars (32 bytes)")

    # Exercise 2: Sign and Verify
    print("\n📝 EXERCISE 2: Sign and Verify a Transaction")
    print("-"*40)

    # Create a transaction
    transaction = {
        'from': wallet1.public_key_hex[:16],
        'to': wallet2.public_key_hex[:16],
        'amount': '100',
        'asset': 'XRP'
```

```
}

print(f"\nTransaction: {json.dumps(transaction, indent=2)}")

# Sign with wallet1
signed = wallet1.sign_transaction(transaction)
print(f"\nSignature: {signed['signature'][:30]}...")

# Verify with wallet1's public key
print("\nVerifying with correct public key:")
wallet1.verify_signature(transaction, signed['signature'], wallet1.public_key_hex)

# Try to verify with wrong public key
print("\nVerifying with WRONG public key (Bob's key):")
wallet1.verify_signature(transaction, signed['signature'], wallet2.public_key_hex)

# Exercise 3: Create Payment Condition
print("\n📝 EXERCISE 3: Create ILP Payment Condition")
print("-"*40)

condition = wallet1.create_payment_condition(wallet2.public_key_hex, "1000")
print(f"\nPayment Condition: {condition['condition'][:30]}...")
print(f"Payment Details: {condition['details']}")

# Exercise 4: Tamper Detection
print("\n📝 EXERCISE 4: Tamper Detection")
print("-"*40)

original_tx = "send 100 XRP to Bob"
print(f"Original: {original_tx}")
```

```
    # Sign original
    signed = wallet1.sign_transaction(original_tx)

    # Try to verify tampered message
    tampered_tx = "send 1000 XRP to Bob"
    print(f"Tampered: {tampered_tx}")

    print("\nVerifying tampered message:")
    wallet1.verify_signature(tampered_tx, signed['signature'], wallet1.public_key_hex)

if __name__ == "__main__":
    run_lab2()
```

Lab 2 Questions

1. Why can't someone forge a signature without the private key?

2. What happens if the transaction data is modified after signing?

3. How would you implement key rotation for a long-lived wallet?

🔬 Lab 3: Multi-Signature Conditions

Theory

Multi-signature conditions require multiple parties to sign before funds can be released. This is crucial for high-value accounts and organizational wallets.

```
Condition = Require(M-of-N Signatures)
Fulfillment = Collect M valid signatures → Transaction executes
```

Complete Code

```python
"""
Lab 3: Multi-Signature Conditions
Implement 2-of-3 multi-signature wallet for ILP
"""

import nacl.signing
import nacl.encoding
import hashlib
import base64
import json
import time
from collections import defaultdict

class MultiSigWallet:
    """
```

```
2-of-3 Multi-signature wallet implementation
"""

def __init__(self, wallet_id, required_signatures=2):
    self.wallet_id = wallet_id
    self.required = required_signatures
    self.signers = []  # List of signer dictionaries
    self.approvals = defaultdict(dict)  # Track approvals per transaction

def add_signer(self, name, public_key_hex):
    """
    Add a signer to the wallet

    Args:
        name: Signer's name/identifier
        public_key_hex: Signer's public key in hex
    """
    self.signers.append({
        'name': name,
        'public_key': public_key_hex
    })
    print(f"✅ Added signer: {name}")

def create_multisig_transaction(self, destination, amount):
    """
    Create a transaction requiring multiple signatures

    Args:
        destination: Destination address
        amount: Transaction amount
```

```
Returns:
    dict: Transaction details and condition
"""
# Generate transaction ID
tx_id = hashlib.sha256(
    f"{destination}{amount}{time.time()}".encode()
).hexdigest()[:16]

transaction = {
    'id': tx_id,
    'destination': destination,
    'amount': amount,
    'timestamp': time.time(),
    'required_signatures': self.required,
    'signers': [s['public_key'] for s in self.signers],
    'approvals': []  # Will store signatures
}

# Create condition (hash of transaction)
tx_bytes = json.dumps(transaction, sort_keys=True).encode()
condition = hashlib.sha256(tx_bytes).digest()
condition_b64 = base64.urlsafe_b64encode(condition).decode('utf-8')

self.approvals[tx_id] = {
    'transaction': transaction,
    'condition': condition_b64,
    'signatures': []
}
```

```
    print(f"\n📝 Created transaction: {tx_id}")
    print(f"📌 Condition: {condition_b64[:30]}...")
    print(f"👥 Required signatures: {self.required}")

    return {
        'tx_id': tx_id,
        'condition': condition_b64,
        'transaction': transaction
    }

def sign_transaction(self, tx_id, signer_private_key_hex, signer_name):
    """
    Sign a transaction (simulated - in production use proper crypto)

    Args:
        tx_id: Transaction ID
        signer_private_key_hex: Signer's private key
        signer_name: Name of signer

    Returns:
        dict: Signature result
    """
    if tx_id not in self.approvals:
        return {'error': 'Transaction not found'}

    # Simulate signing (in reality, would use proper Ed25519)
    signature = hashlib.sha256(
        f"{tx_id}{signer_private_key_hex}{time.time()}".encode()
    ).hexdigest()[:32]
```

```
    # Store approval
    self.approvals[tx_id]['signatures'].append({
        'signer': signer_name,
        'signature': signature,
        'timestamp': time.time()
    })

    current = len(self.approvals[tx_id]['signatures'])
    print(f"\n✍ {signer_name} signed transaction {tx_id}")
    print(f"   Signatures: {current}/{self.required}")

    return {
        'tx_id': tx_id,
        'signature': signature,
        'current_signatures': current,
        'required': self.required
    }

def execute_transaction(self, tx_id):
    """
    Execute transaction if enough signatures collected

    Args:
        tx_id: Transaction ID

    Returns:
        dict: Execution result
    """
    if tx_id not in self.approvals:
        return {'success': False, 'error': 'Transaction not found'}
```

```
    tx_data = self.approvals[tx_id]
    signatures = tx_data['signatures']

    if len(signatures) >= self.required:
        print(f"\n✓ TRANSACTION EXECUTED: {tx_id}")
        print(f"   Destination: {tx_data['transaction']['destination']}")
        print(f"   Amount: {tx_data['transaction']['amount']}")
        print(f"   Signatures provided: {len(signatures)}")

        return {
            'success': True,
            'transaction': tx_data['transaction'],
            'signatures': signatures
        }
    else:
        print(f"\n□ Insufficient signatures: {len(signatures)}/{self.required}")
        return {
            'success': False,
            'error': 'Insufficient signatures',
            'current': len(signatures),
            'required': self.required
        }

def get_fulfillment(self, tx_id):
    """
    Get fulfillment data for a completed transaction

    Args:
        tx_id: Transaction ID
```

```
        Returns:
            dict: Fulfillment data or None
        """
        if tx_id not in self.approvals:
            return None

        tx_data = self.approvals[tx_id]
        if len(tx_data['signatures']) < self.required:
            return None

        fulfillment = {
            'condition': tx_data['condition'],
            'signatures': [
                {'signer': s['signer'], 'signature': s['signature']}
                for s in tx_data['signatures'][:self.required]
            ],
            'transaction': tx_data['transaction']
        }

        return fulfillment

def run_lab3():
    """Main function to run Lab 3 exercises"""
    print("\n" + "="*60)
    print("LAB 3: MULTI-SIGNATURE CONDITIONS")
    print("="*60)

    # Create a 2-of-3 multisig wallet
    company_wallet = MultiSigWallet("CompanyFunds", required_signatures=2)
```

```
# Add three signers
company_wallet.add_signer("Alice", "pubkey_alice_1234567890abcdef")
company_wallet.add_signer("Bob", "pubkey_bob_1234567890abcdef")
company_wallet.add_signer("Charlie", "pubkey_charlie_1234567890abcdef")

# Create a transaction
print("\n" + "-"*40)
print("Creating large payment requiring approval")
tx = company_wallet.create_multisig_transaction(
    destination="vendor_wallet_xyz",
    amount="10000"
)

# Try to execute with insufficient signatures
print("\n" + "-"*40)
print("Attempt 1: Only Alice signs")
company_wallet.sign_transaction(tx['tx_id'], "alice_privkey_123", "Alice")
company_wallet.execute_transaction(tx['tx_id'])

# Add second signature
print("\n" + "-"*40)
print("Attempt 2: Bob also signs")
company_wallet.sign_transaction(tx['tx_id'], "bob_privkey_456", "Bob")
result = company_wallet.execute_transaction(tx['tx_id'])

# Get fulfillment
if result['success']:
    fulfillment = company_wallet.get_fulfillment(tx['tx_id'])
    print("\n🎁 Fulfillment data ready for ILP:")
```

```
        print(json.dumps(fulfillment, indent=2))

    # Exercise: Test different combinations
    print("\n📝 EXERCISE: Test Different Scenarios")
    print("-"*40)

    print("1. What if only 1 of 3 signs? - Transaction stays locked")
    print("2. What if Alice and Charlie sign? - Transaction executes")
    print("3. What if all three sign? - Still executes (only need 2)")

if __name__ == "__main__":
    run_lab3()
```

Lab 3 Questions

1. Why would an organization use a 3-of-5 multisig instead of 2-of-3?

2. How does multisig protect against a single compromised key?

3. What happens to the condition if the transaction details change?

🔬 Lab 4: Secure Key Storage

Theory

In production, private keys must never be hardcoded. They should be stored in environment variables, encrypted files, or hardware security modules (HSMs).

```
❌ BAD:  private_key = "abc123..."  # Hardcoded in source

✅ GOOD: private_key = os.getenv("WALLET_PRIVATE_KEY")  # Environment variable

✅ BETTER: Decrypt from encrypted file with password
```

Complete Code

```python
"""
Lab 4: Secure Key Storage and Environment Configuration

Learn how to securely store and access wallet keys
"""

import os
import base64
import json
import hashlib
from pathlib import Path
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2
import getpass
```

```
class SecureWalletStorage:
    """
    Demonstrates secure key storage practices
    """

    def __init__(self, wallet_name):
        self.wallet_name = wallet_name
        self.env_prefix = f"ILP_WALLET_{wallet_name.upper()}"

    def generate_wallet_keys(self):
        """
        Generate new wallet keys using Ed25519

        Returns:
            dict: Contains private and public keys
        """
        import nacl.signing
        import nacl.encoding

        signing_key = nacl.signing.SigningKey.generate()
        private_key = signing_key.encode(encoder=nacl.encoding.HexEncoder).decode('utf-8')
        public_key = signing_key.verify_key.encode(encoder=nacl.encoding.HexEncoder).decode('utf-8')

        return {
            'private_key': private_key,
            'public_key': public_key
        }

    def show_env_config(self):
```

```
    """
    Show how to configure environment variables
    """
    keys = self.generate_wallet_keys()

    print("\n" + "="*60)
    print("ENVIRONMENT VARIABLE CONFIGURATION")
    print("="*60)
    print("\nSet these environment variables (never hardcode!):")
    print("-"*40)
    print(f"export {self.env_prefix}_PRIVATE_KEY={keys['private_key']}")
    print(f"export {self.env_prefix}_PUBLIC_KEY={keys['public_key']}")
    print(f"export {self.env_prefix}_SEED={os.urandom(16).hex()}")

    print("\nIn Python, access them like this:")
    print("-"*40)
    print("import os")
    print(f"private_key = os.getenv('{self.env_prefix}_PRIVATE_KEY')")
    print(f"public_key = os.getenv('{self.env_prefix}_PUBLIC_KEY')")

def load_from_env(self):
    """
    Load keys from environment variables

    Returns:
        dict: Keys if found, None otherwise
    """
    private_key = os.getenv(f"{self.env_prefix}_PRIVATE_KEY")
    public_key = os.getenv(f"{self.env_prefix}_PUBLIC_KEY")
```

```
    if not private_key or not public_key:
        print(f"\n⚠️  Environment variables not set for {self.wallet_name}")
        print(f"   Please set: {self.env_prefix}_PRIVATE_KEY and {self.env_prefix}_PUBLIC_KEY")
        return None

    return {
        'private_key': private_key,
        'public_key': public_key
    }

def encrypt_wallet_file(self, password=None):
    """
    Encrypt wallet keys to a file

    Args:
        password: Encryption password (if None, prompt user)

    Returns:
        str: Filename of encrypted wallet
    """
    if not password:
        password = getpass.getpass("Enter encryption password: ")

    # Generate keys
    keys = self.generate_wallet_keys()
    wallet_data = json.dumps(keys).encode()

    # Generate key from password using PBKDF2
    salt = os.urandom(16)
    kdf = PBKDF2(
```

```
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=100000,
        )
        key = base64.urlsafe_b64encode(kdf.derive(password.encode()))
        f = Fernet(key)

        # Encrypt
        encrypted_data = f.encrypt(wallet_data)

        # Save to file
        filename = f"{self.wallet_name}.encrypted_wallet"
        with open(filename, 'wb') as f:
            f.write(salt + encrypted_data)

        print(f"\n✓ Wallet encrypted and saved to {filename}")
        print(f"🔑 File size: {len(encrypted_data) + 16} bytes")
        print(f"⚠  Remember your password! No recovery possible.")

        return filename

    def decrypt_wallet_file(self, filename, password=None):
        """
        Decrypt wallet from file

        Args:
            filename: Path to encrypted wallet file
            password: Decryption password (if None, prompt user)
```

```
    Returns:
        dict: Decrypted keys or None if failed
    """
    if not password:
        password = getpass.getpass("Enter decryption password: ")

    # Read file
    with open(filename, 'rb') as f:
        salt = f.read(16)
        encrypted_data = f.read()

    # Derive key using same parameters
    kdf = PBKDF2(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
    )
    key = base64.urlsafe_b64encode(kdf.derive(password.encode()))
    f = Fernet(key)

    try:
        # Decrypt
        decrypted = f.decrypt(encrypted_data)
        keys = json.loads(decrypted)
        print(f"\n✓ Wallet decrypted successfully!")
        return keys
    except Exception as e:
        print(f"\n✗ Decryption failed: {e}")
        return None
```

```
    def demonstrate_secure_memory(self):
        """
        Demonstrate secure memory handling concepts
        """
        print("\n" + "="*60)
        print("SECURE MEMORY HANDLING")
        print("="*60)
        print("""
```

In production ILP implementations:

1. Use memory locking to prevent swapping to disk

2. Zeroize keys immediately after use

3. Never log private keys or seeds

4. Use redacted debug output

```
        """)

        # Example of redacted output
        class RedactedKey:
            def __init__(self, key):
                self.key = key

            def __repr__(self):
                return "<redacted_private_key>"

            def get_for_signing(self):
                # Only expose when needed
                return self.key

        sample_key = "secret_key_1234567890abcdef"
        redacted = RedactedKey(sample_key)
```

```
        print(f"\nNormal print: {sample_key}")
        print(f"Redacted print: {redacted}")

    def practice_exercise(self):
        """
        Interactive exercise for students
        """
        print("\n" + "="*60)
        print("PRACTICE EXERCISE")
        print("="*60)

        print("\nScenario: You're deploying an ILP connector")

        print("\nTask 1: Environment Variables")
        print("Create a .env file with:")
        print("ILP_WALLET_PROD_PRIVATE_KEY=your_private_key_here")
        print("ILP_WALLET_PROD_PUBLIC_KEY=your_public_key_here")

        print("\nTask 2: Write secure loading code")
        print("""
import os
from dotenv import load_dotenv

load_dotenv()  # Load .env file

def get_wallet_keys(wallet_name):
    private = os.getenv(f"ILP_WALLET_{wallet_name}_PRIVATE_KEY")
    public = os.getenv(f"ILP_WALLET_{wallet_name}_PUBLIC_KEY")

    if not private or not public:
```

```
        raise ValueError("Keys not found in environment")

    return private, public
        """)

        print("\nTask 3: Answer these questions")
        print("1. Why shouldn't you commit .env files to git?")
        print("2. How would you rotate keys in production?")
        print("3. What's the difference between env vars and encrypted files?")

def run_lab4():
    """Main function to run Lab 4 exercises"""
    print("\n" + "="*60)
    print("LAB 4: SECURE KEY STORAGE")
    print("="*60)

    # Demo 1: Environment Variables
    storage = SecureWalletStorage("DEMO")
    storage.show_env_config()

    # Demo 2: Encrypted File Storage
    print("\n" + "-"*40)
    print("ENCRYPTED FILE STORAGE DEMO")
    print("-"*40)

    # Create encrypted wallet
    filename = storage.encrypt_wallet_file(password="demo_password123")

    # Decrypt and verify
    print("\nAttempting to decrypt...")
```

```
    decrypted = storage.decrypt_wallet_file(filename, password="demo_password123")
    if decrypted:
        print(f"Recovered public key: {decrypted['public_key'][:32]}...")

    # Demo 3: Secure Memory
    storage.demonstrate_secure_memory()

    # Demo 4: Interactive Practice
    storage.practice_exercise()

if __name__ == "__main__":
    run_lab4()
```

Lab 4 Questions

1. Why are environment variables better than hardcoded keys?

2. What additional security does encrypted file storage provide?

3. How would you implement key rotation without downtime?

4. Why is it important to zeroize keys from memory?

Security Best Practices Checklist

Key Management

- Never hardcode keys in source code
- Use environment variables for development
- Use encrypted files or HSMs for production
- Implement key rotation procedures
- Use separate keys for different environments

Operational Security

- Never log private keys or seeds
- Use locked memory to prevent swapping
- Implement proper access controls
- Regular security audits

ILP-Specific

- Validate all conditions before fulfilling
- Check for replay attacks
- Monitor for unusual fulfillment patterns

References

Official Specifications

1. [Crypto-Conditions RFC](https://tools.ietf.org/html/draft-thomas-crypto-conditions-03)
2. [Interledger Protocol Specifications](https://github.com/interledger/rfcs)

Libraries

3. [PyNaCl Documentation](https://pynacl.readthedocs.io/)
4. [Cryptography Library Docs](https://cryptography.io/)