Introduction

This report documents the design and implementation of a secure Distributed Ledger Technology (DLT)-based inventory management system. The primary objective is an extension of the fundamentals implemented in Assignment 1 by integrating advanced cryptographic mechanisms and consensus protocols to ensure data integrity, authenticity, and confidentiality across a network of inventory nodes.

The system simulates a private blockchain environment where four inventory nodes (A, B, C, D) collaboratively manage and verify inventory records. Key functionalities include RSA-based digital signatures for record authentication, Practical Byzantine Fault Tolerance (PBFT) for consensus, and Harn's identity-based multi-signature scheme for secure query verification. Additionally, the system ensures that sensitive data is encrypted before transmission to authorised users.

Implementation Details

Technologies and Tools

- Programming Language: Python
- Web Framework: Flask
- Cryptographic Libraries: Python's built-in hashlib for hashing and pow functions for RSA operations such as modular exponentiation.
- Data Storage: JSON files simulating individual node databases
- Consensus Protocol: Practical Byzantine Fault Tolerance (PBFT)
- Multi-Signature Scheme: Harn's Identity-Based Multi-Signature (PBFT CONSENSUS PROTOCOL IN CRYPTO - CavemenTech)

PART 1, TASK 1 -

System Architecture

The system comprises four inventory nodes, each possessing unique RSA key pairs. These nodes operate in a permissioned network, allowing them to authenticate records, participate in consensus, and respond to queries securely. A Flask-based web interface facilitates user interactions, including record submissions and inventory queries.

Hard-coded configuration:

```python
class NodeConfig:
    def __init__(self, p, q, e):
        self.p = p
        self.q = q
        self.e = e
        self.n = p * q

        def get_public_key(self):
            return (self.e, self.n)

# Simulated nodes configuration
NODES = {
    "A": NodeConfig(
        p=121061376573514731110693631186659397807993870 7,
        q=124784285028203575361595134796443724819023186 3,
        e=815459040813953176289801
    ),
    "B": NodeConfig(
        p=787435686772982288169641922308628444877260947,
        q=132530523388609605331034041846738539723937537 9,
        e=692450682143089563609787
    ),
    "C": NodeConfig(
        p=101424730099103944486420151827501824036120511 1,
        q=904030450302158058469475048755214591704639633,
        e=11587494220150353884380 57
    ),
    "D": NodeConfig(
        p=128773720089142562133855102076285871028163831 7,
        q=133090912572507346979495323415152520108453760 7,
        e=33981230465225879849295979
    )
}
```

```python
TOTAL_NODES = 4
CONSENSUS_PROTOCOL = "PBFT"
MAX_FAULTY_NODES = 1  # f=1 for 4 nodes # Ref: https://www.geeksforgeeks.org/minimum
REQUIRED_APPROVALS = 2 * MAX_FAULTY_NODES + 1  # 3 for f=1
CONSENSUS_THRESHOLD = .75  # Honest Nodes ≥ (Total Nodes / 3) * 2 --> (4/3) * 2 -->
```

RSA Digital Signature Implementation

Key Generation:

Each node generates its RSA key pair using predefined prime numbers p and q, and a public exponent e. The modulus n and private exponent d are computed as follows:

self.n = p * q

self.phi = (p - 1) * (q - 1)

self.d = pow(e, -1, self.phi)

```python
class RSANode:
    def __init__(self, name, p, q, e):
        self.name = name
        self.p = p
        self.q = q
        self.e = e
        self.n = p * q
        self.phi = (p - 1) * (q - 1)
        self.d = pow(e, -1, self.phi)
```

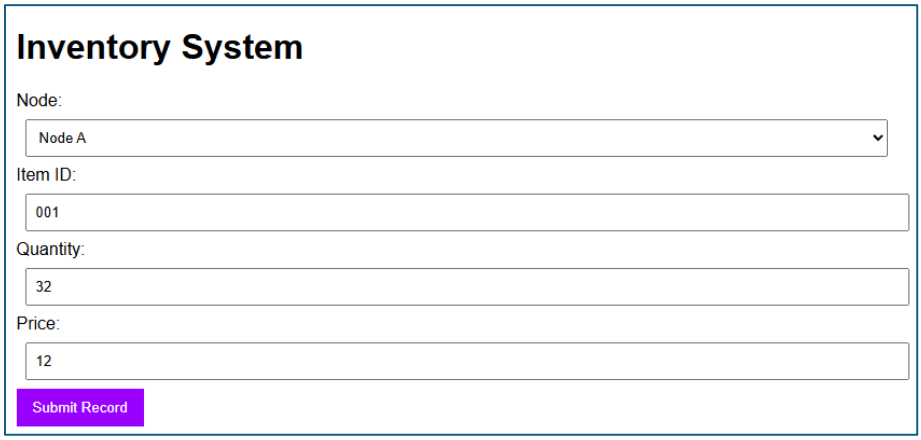
Signing Process:

When a node wishes to add a new inventory record, it creates a string combining its name, item ID, quantity, and price. This string is hashed using SHA-256, and the hash is signed using the node's private key:

message_bytes = message.encode('utf-8')

h = int.from_bytes(hashlib.sha256(message_bytes).digest(), 'big')

signature = pow(h, self.d, self.n)



Record:

```python
# Create and sign record
record = f"{node}:{item}:{qty}:{price}"
print(f"\nSigning record: '{record}'")
signature = nodes[node].sign(record)
print(f"Generated signature: {signature}")
```

Output:

```
Signing record: 'A:001:32:12'
Generated signature: 459443389056899542787671227362780127535113069883375530171213086314929178111452755725588580
```

Signature process:

```python
def sign(self, message):
    message_bytes = message.encode('utf-8')
    h = int.from_bytes(hashlib.sha256(message_bytes).digest(), 'big')

    # Ensure h < n (RSA requirement)
    if h >= self.n:
        h = h % self.n

    return pow(h, self.d, self.n) # return signature val
```

Verification Process:

Other nodes verify the signature by computing the hash of the received message and comparing it with the decrypted signature using the signer's public key:

h_original = int.from_bytes(hashlib.sha256(message_bytes).digest(), 'big')

h_recovered = pow(signature, signer_e, signer_n)

valid = h_original == h_recovered

```python
def verify(self, message, signature, signer_name):
    # Get the signer's public key from  config
    signer = NODES[signer_name]
    signer_e, signer_n = signer.e, signer.n


    # Compute original hash
    message_bytes = message.encode('utf-8')
    h_original = int.from_bytes(hashlib.sha256(message_bytes).digest(), 'big')

    # Convert signature to int if needed
    sig_int = int(signature) if isinstance(signature, str) else signature

    # Recover hash using signer's public key
    h_recovered = pow(sig_int, signer_e, signer_n)

    print(f"\nVerification by {self.name} for {signer_name}:")
    print(f"Using public key (e,n): ({signer_e}, {signer_n})")
    print(f"Original hash: {h_original}")
    print(f"Recovered hash: {h_recovered}")
    print(f"Hash match: {h_original == h_recovered}")

    return {
    'valid': h_original == h_recovered,
    'original_hash': h_original,
    'recovered_hash': h_recovered,
    'e': signer_e,
    'n': signer_n
}
```

“h_original” is the computation of the SHA-256 hash of the original message in big integer format.

“h_recovered” is recovers the hash from the signature using the signer's public key (n, e).

In the return statement, “‘valid’: h_original == h_recovered” checks if the recovered hash matches the original hash, if they match the signature is valid because the message was not altered as it is the same hash and the signature was produced by the holder of the private key corresponding to the public key(n, e).

Note: the print(f... statements are for debugging in the VS code terminal.

Example of output from Record – A:001:32:12

```
Signing record: 'A:001:32:12'
Generated signature: 459443389056899542787671227362780127535113069883375530171213086314929178111452755725588580

Verification by B for A:
Using public key (e,n): (815459040813953176289801, 151065573202561493161847311349093693600701087608649273042221881743560791950248623915842114 1)
Original hash: 109365136687315522802571383084026243949449443893876505797005808241929833635567
Recovered hash: 109365136687315522802571383084026243949449443893876505797005808241929833635567
Hash match: True
```

Record Status: COMMITTED

Record: A:001:32:12

Signature:
459443389056899542787671227362780127535113069883375530171213086314929178111452755725588580

Consensus: 4 / 3 required approvals

Verification Results

Node B: Verified

Public Key (e,n): (815459040813953176289801,
151065573202561493161847311349093693600701087608649273042221881743560791950248623915842114 1)

Original Hash:
109365136687315522802571383084026243949449443893876505797005808241929833635567

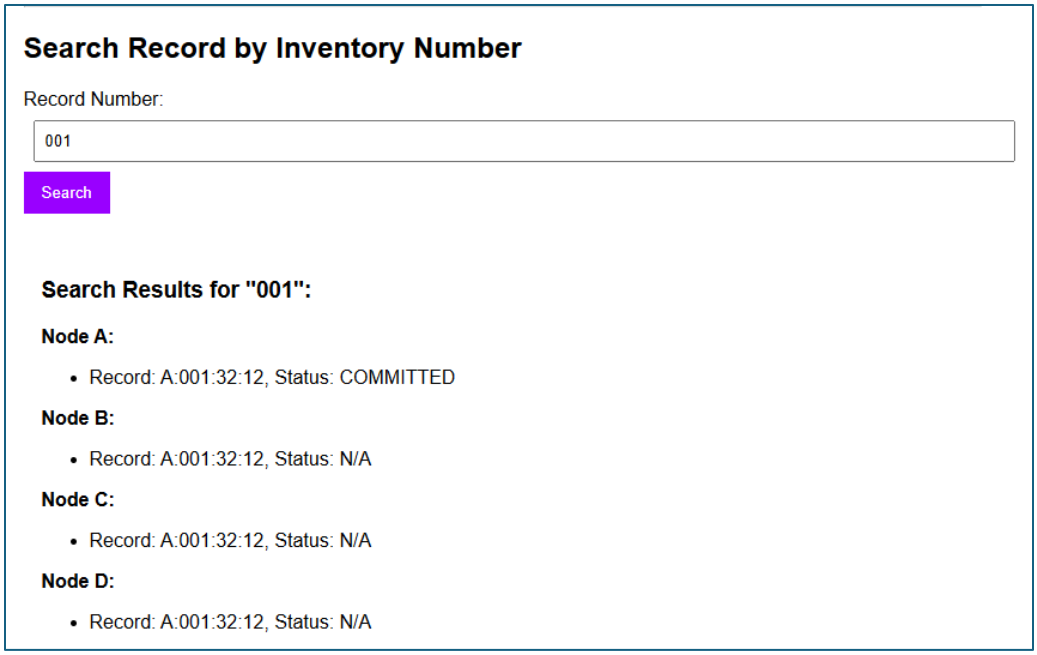
Recovered Hash:
109365136687315522802571383084026243949449443893876505797005808241929833635567

Hashes match

Consensus: Approved

Threshold: 75%

Search function (UI):



Code for search function (search_record flask endpoint)

```python
@app.route('/search_record', methods=['GET'])
def search_record():
    record_number = request.args.get('record_number')
    if not record_number:
        return jsonify({"error": "Missing record_number parameter"}), 400

    found_in_nodes = {}

    for node_name in nodes:
        db = get_db(node_name)
        matching_records = [r for r in db.get("records", []) if record_number in r.get("record", "")]
        if matching_records:
            found_in_nodes[node_name] = matching_records

    if not found_in_nodes:
        return jsonify({"message": f"No records found with number '{record_number}'"}), 404

    return jsonify({
        "record_number": record_number,
        "found_in_nodes": found_in_nodes
    })
```

Record_number extracts the record number from URL, which will be used for filtering the relevant records based on the search.

Found_in_nodes = {} is a dictionary which will store the results for each node

Then, there is a loop that calls to get the database and retrieve the records from the node (db.get("records", [])).

If there is a match with a record, it is saved to found_in_nodes[node_name] and the results are returned as json.

Part 2, Task 2

Consensus Protocol

Chosen Protocol: Practical Byzantine Fault Tolerance (PBFT)

The Practical Byzantine Fault Tolerance (PBFT) consensus protocol was selected for this system due to its alignment with private (permissioned) blockchain architectures, where all participating nodes are known and trusted in advance and the number of nodes in the network are small (RMIT LECTURE 4). This matches our scenario, as the inventory management system operates within a fixed network of validated nodes (A, B, C, D). PBFT's design offers critical advantages over alternatives like Proof-of-Work (PoW) and Proof-of-Stake (PoS), particularly in terms of efficiency and fault tolerance.

Unlike PoW—which is used in public blockchains like Bitcoin and allows unknown participants to join—PBFT eliminates the need for computationally intensive mining. PoW requires miners to solve complex mathematical puzzles to validate transactions, resulting in excessive energy consumption and delays (Investopedia, 2024). Similarly, PoS systems introduce complexity through staking mechanics and remain vulnerable to attacks like "nothing-at-stake," where validators might approve conflicting transactions (NIST, n.d). Both PoW and PoS also suffer from probabilistic finality, meaning transactions can be temporarily reverted due to forks, whereas PBFT guarantees immediate and irreversible consensus once a record is committed.

PBFT's efficiency stems from its streamlined three-phase process (Pre-Prepare, Prepare, Commit), which enables a very fast consensus while tolerating malicious nodes. This makes it ideal for real-time systems like inventory tracking, where delays or forks are unacceptable. Additionally, PBFT avoids the resource waste of PoW/PoS by relying on cryptographic signatures (RSA) and a 2F+1 (where F is positive verification) approval threshold instead of mining or staking.

The PBFT protocol is designed to maintain consensus in distributed systems even when some nodes are faulty or malicious. The system's resilience is determined by the formula n (total nodes) = 3f + 1 (f = maximum number of faulty nodes tolerated). This inventory has four nodes (n=4), solving for f (4 = 3f + 1 → f=1) demonstrates the network can withstand one faulty node while maintaining correct operation (GeeksforGeeks, 2024).

PBFT achieves this fault tolerance through its three-phase consensus mechanism:

(1) the pre-prepare phase where the leader node proposes a value

(2) the prepare phase where nodes validate and broadcast this proposal,

(3) the commit phase where nodes finalise the agreement after receiving sufficient confirmations.

Chosen Protocol: Practical Byzantine Fault Tolerance (PBFT)

For this blockchain-based inventory system, the **Practical Byzantine Fault Tolerance (PBFT)** consensus protocol was selected. PBFT is particularly suitable for permissioned networks with known participants, such as the multiple inventory nodes in this system.

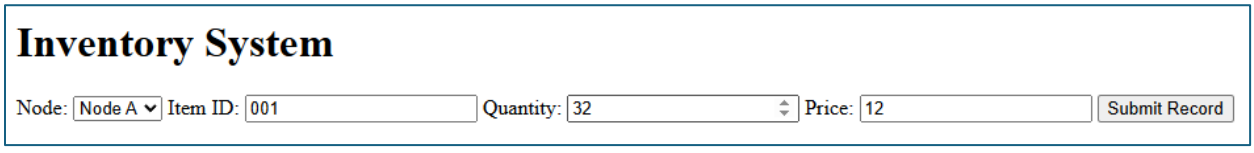
- **Rationale:**
    - The system involves a fixed set of trusted inventory nodes (A, B, C, D), making PBFT ideal because it efficiently tolerates faulty or malicious nodes up to a certain threshold.
    - PBFT does not require resource-intensive computations like Proof of Work (PoW), making it faster and more energy-efficient for the scale of this project.
    - The protocol enables fast finality on consensus, allowing nodes to agree on the acceptance of new inventory records quickly.
    - The network does not involve a token or coin, making Proof of Stake (PoS) or similar protocols irrelevant.

Therefore, PBFT offers the best trade-off between security, performance, and suitability for this permissioned inventory network.

Record Submission (index.html)

A new block is added by a node from the webpage (simulation) to the database (json).

The example below shows a successful block being added to the permissioned blockchain system.



(1) Pre-prepare

Trigger: A node submits a record via the /submit endpoint (e.g., A:001:32:12).

```python
# Route for submitting a record
@app.route('/submit', methods=['POST'])
def submit():
    global global_sequence_number
    data = request.json
    node = data.get("node")
    record = data.get("record")


    if not node or not record or node not in nodes:
        return jsonify({"error": "Invalid input"}), 400

    # Check if this node is the primary for the current view
    current_view = nodes[node].view_number
    is_primary = (node == get_primary_node(current_view))
```

Action: The primary node (determined by view_number % N) signs the record and broadcasts a pre-prepare message.

```python
def is_primary_node(node_name, view_number):
    return node_name == list(nodes.keys())[view_number % len(nodes)]
```

In the /submit endpoint:

```python
# Check if this node is the primary for the current view
current_view = nodes[node].view_number
is_primary = (node == get_primary_node(current_view))


 # Use global sequence number and increment it
global_sequence_number += 1
sequence_number = global_sequence_number

# Update the node's sequence number to match (for consistency)
nodes[node].sequence_number = sequence_number

# --- Phase 1: Pre-Prepare ---

signature = nodes[node].sign(record)

# Store pre-prepare message
pre_prepare = {
    'sequence': sequence_number,
    'view': current_view,
    'phase': 'pre-prepare',
    'record': record,
    'signature': signature,
    'sender': node,
    'is_primary': is_primary
}
nodes[node].message_log.append(pre_prepare)
```

The primary node (A) signs the record and broadcasts the pre-prepare message to other nodes.

```python
def sign(self, message):
    message_bytes = message.encode()
    h = int.from_bytes(hashlib.sha256(message_bytes).digest(), 'big')
    if h >= self.n:
        h = h % self.n
    return pow(h, self.d, self.n)
```

(2) Prepare

Goal: Non-primary nodes verify the pre-prepare message and broadcast prepare messages.

Validation: Each replica checks:

Global variables:

```python
global_sequence_number = 0
inventory_ledger = []
```

global_sequence_number: Keeps track of the ordering of records.

inventory_ledger: A local list holding committed records.

The signature is valid (using RSANode.verify()).

RSANode:

```python
class RSANode:
    def __init__(self, name, p, q, e):
        self.name = name
        self.p = p
        self.q = q
        self.e = e
        self.n = p * q
        self.phi = (p - 1) * (q - 1)
        self.d = pow(e, -1, self.phi)
        self.view_number = 0
        self.sequence_number = 0
        self.prepare_messages = {}
        self.commit_messages = {}
        self.message_log = []
```

Initialise all nodes from the predefined configuration (config.py).

```python
nodes = {name: RSANode(name, params.p, params.q, params.e) for name, params in NODES.items()}
```

Validation:

```python
def verify(self, message, signature, signer_name):
    signer = nodes[signer_name]
    signer_e, signer_n = signer.e, signer.n
    message_bytes = message.encode()
    h_original = int.from_bytes(hashlib.sha256(message_bytes).digest(), 'big')
    sig_int = int(signature) if isinstance(signature, str) else signature
    h_recovered = pow(sig_int, signer_e, signer_n)
    return h_original == h_recovered
```

The message is then prepared for each

```python
# --- Phase 2: Prepare ---
prepare_messages = []
for name in nodes:
    if name == node:
        continue

    # Each replica verifies the pre-prepare
    if not nodes[name].verify(record, signature, node):
        continue

    prepare = {
        'sequence': sequence_number,
        'view': current_view,
        'phase': 'prepare',
        'record': record,
        'signature': nodes[name].sign(f"{sequence_number}:{current_view}:{record}"),
        'sender': name
    }
    nodes[name].prepare_messages[(sequence_number, current_view)] = prepare
    nodes[name].message_log.append(prepare)
    prepare_messages.append(prepare)
```

Prepare_messages stores all valid prepare messages from replicas.

Prepare structure:

- sequence: Unique ID for the consensus round (matches pre-prepare).
- view: Current view number (to prevent replay attacks).
- phase: Identifies this as a prepare message.
- record: The original record (e.g., A:001:32:12).
- signature: The replica's digital signature over sequence:view:record.
- sender: The node's ID (e.g., B).

Nodes B, C, and D validate the pre-prepare and broadcast prepare messages.

(3) Commit

Condition: Consensus is reached if 2f + 1 nodes (3/4) agree. (1 faulty node)

Action: Nodes broadcast commit messages and append the record to their databases.

Prepare_messages is used to check if it meets the threshold for the number of required approvals.

```
# --- Phase 3: Commit ---
commit_messages = []
consensus_reached = False

if len(prepare_messages) + 1 >= REQUIRED_APPROVALS:  # +1 for primary
```

commit_messages: Stores all valid commit messages from nodes.

consensus_reached: Flag to track if consensus is achieved.

For N=4 and REQUIRED_APPROVALS=3, at least 2 replicas must send prepare messages.

```
for name in nodes:
    commit = {
        'sequence': sequence_number,
        'view': current_view,
        'phase': 'commit',
        'record': record,
        'signature': nodes[name].sign(f"commit:{sequence_number}:{current_view}:{record}"),
        'sender': name
    }
    nodes[name].commit_messages[(sequence_number, current_view)] = commit
    nodes[name].message_log.append(commit)
    commit_messages.append(commit)
```

Each node signs a string formatted as commit:{sequence}:{view}:{record} (e.g., commit:1:0:A:001:32:12). This cryptographically binds the node to the decision.

Local Storage: Stores the commit in commit_messages (node-specific) and message_log (audit trail).

Broadcast: Adds the message to commit_messages for threshold checking. Replicas only proceed if the pre-prepare is valid (prevents malicious primary nodes).

```
if len(commit_messages) + 1 >= REQUIRED_APPROVALS:  # +1 for primary
    status = "committed"
    # Apply to all nodes' databases
    for name in nodes:
        db = get_db(name)
        db["records"].append({
            "record": record,
            "signature": str(signature),
            "status": "committed",
            "verified_by": "PBFT",
            "sequence": sequence_number,
            "view": current_view,
            "timestamp": datetime.datetime.now().isoformat(),
            "is_primary": is_primary
        })
```

The record is appended to the JSON database. The consensus threshold is verified

```
save_db(name, db)
inventory_ledger.append({
    "record": record,
    "signature": str(signature),
    "status": "committed",
    "verified_by": "PBFT",
    "sequence": sequence_number,
    "view": current_view,
    "timestamp": datetime.datetime.now().isoformat(),
    "is_primary": is_primary
})
```

Sync global ledger

```
else:
    status = "pending"
```

If consensus is achieved:

If commit_messages + primary (+1) ≥ REQUIRED_APPROVALS, the record is marked committed.

The record is appended to all nodes' databases (node_a.json, etc.) and the inventory_ledger.

If consensus failed:

Status remains pending.

Nodes A, B, and C send commit messages, meeting the threshold (3/4). The record is stored in all databases.

This structured approach ensures all honest nodes reach consistent agreement on the system state despite potential malicious behaviour from up to f (1) nodes

```
    return jsonify({
        "status": f"Consensus {status}",
        "record_status": status,
        "record": record,
        "signature": str(signature),
        "sequence": sequence_number,
        "view": current_view,
        "prepares_count": len(prepare_messages),
        "commits_count": len(commit_messages),
        "prepares": prepare_messages,
        "commits": commit_messages,
        "is_primary": is_primary,
        "consensus_reached": len(commit_messages) + 1 >= REQUIRED_APPROVALS,
```

It provides a detailed summary of the consensus outcome, as a structured JSON response to the front end including cryptographic proofs and participation details.

Part 2, Task 3

Key Setup:

A Private Key Generator (PKG) computes each node's private key based on Harn's identity-based multi-signature scheme. This approach simplifies public key management by deriving keys from unique identities (IACR, n.d.)

Updated config file:

```
class PKGConfig:
    def __init__(self):
        # PKG Master Key Parameters
        self.p = 10041620364614886393385970004667051792532267 03
        self.q = 95013374115126752211625238592794061826410362 3
        self.e = 973028207197278907211
        self.n = self.p * self.q
        self.phi_n = (self.p - 1) * (self.q - 1)
        self.d = pow(self.e, -1, self.phi_n)  # Private key
```

Master key parameters (p, q, e) are initialized in config.py

Computes n = p*q and private key $d = e^{-1} \bmod \phi(n)$

Generates secret keys for nodes using their identities

```
NODES = {
    "A": NodeConfig(identity=126, random_val=621,
        p=12106137657351473111069363118665939780799387 07,
        q=12478428502820357536159513479644372481902318 63,
        e=815459040813953176289801),
```

- Each node has unique identity, random value, and RSA parameters
- Secret keys are generated using the PKG's master key

```
class ProcurementOfficer:
    def __init__(self):
        self.p = 10809547357224639929883941496028563321006284 17
        self.q = 11581062833200864448909118632998799735422932 43
        self.e = 106506253943651610547613
        self.n = self.p * self.q
        self.phi_n = (self.p - 1) * (self.q - 1)
        self.d = pow(self.e, -1, self.phi_n)  # Private key
```

Query Submission (index.html):

```
document.getElementById('procurement-query-button').addEventListener('click', async () => {
    const itemId = document.getElementById('procurement-item-id').value;
    const resultDiv = document.getElementById('procurement-result');

    try {
        const response = await fetch('/api/verify-query', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ item_id: itemId })
        });

        const data = await response.json();

        if (data.error) {
            resultDiv.innerHTML = `<p class="error">Error: ${data.error}</p>`;
            return;
        }
```

Procurement Officer submits item query via /api/verify-query endpoint

Request includes item ID to search for

```
partial_signatures = []
for node in nodes:
    db = get_db(node)
    for record in db["records"]:
        try:
            if "record" not in record or not isinstance(record["record"], str):
                continue
            parts = record["record"].split(":")
            if len(parts) >= 2 and parts[1] == item_id:
                results.append({
                    "node": parts[0],
                    "item_id": parts[1],
                    "quantity": int(parts[2]) if len(parts) > 2 else None,
                    "price": int(parts[3]) if len(parts) > 3 else None,
                    "signature": record.get("signature")
                })

                partial_sig = HarnMultiSignature.sign_message(node, record["record"])
                partial_signatures.append({
                    "node": node,
                    "partial_signature": str(partial_sig)
                })
```

Each node searches its local database for the item

Generates partial signatures using Harn's algorithm

harnMultiSignature class:

```python
class HarnMultiSignature:
    @staticmethod
    def generate_secret_key(identity):
        return pow(identity, PKG.d, PKG.n)


    @staticmethod
    def sign_message(node_id, message):
        node = NODES[node_id]
        g_i = HarnMultiSignature.generate_secret_key(node.identity)
        r_i = node.random_val
        h = int(hashlib.sha256(message.encode()).hexdigest(), 16) % PKG.n
        return (g_i * pow(r_i, h, PKG.n)) % PKG.n
```

The Response is encrypted using Procurement Officer's public key

Uses RSA encryption with OAEP padding

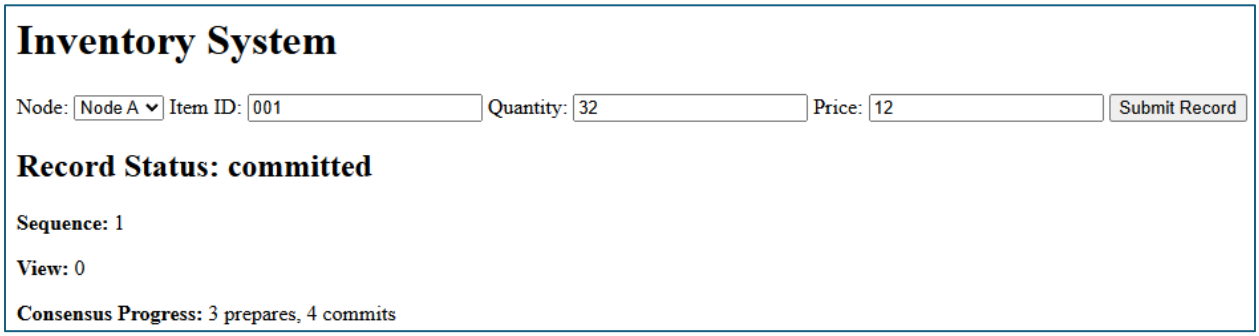
```python
# Encrypt with Procurement Officer's public key
message = json.dumps(response_data).encode()
message_int = int.from_bytes(message, 'big')
if message_int >= PROCUREMENT_OFFICER.n:
    message_int = message_int % PROCUREMENT_OFFICER.n
encrypted = pow(message_int, PROCUREMENT_OFFICER.e, PROCUREMENT_OFFICER.n)

return jsonify({
    "encrypted_response": str(encrypted),
    "verification_parameters": {
        "combined_signature": str(combined_signature),
        "partial_signatures": partial_signatures,
        "pkg_n": str(PKG.n),
        "pkg_e": str(PKG.e)
    }
})
```

Procurement Officer decrypts using private key and verifies the multi-signature

```javascript
document.getElementById('decrypt-button').addEventListener('click', async () => {
  try {
      const decryptRes = await fetch('/api/decrypt', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({
              encrypted: data.encrypted_response
          })
      });
```

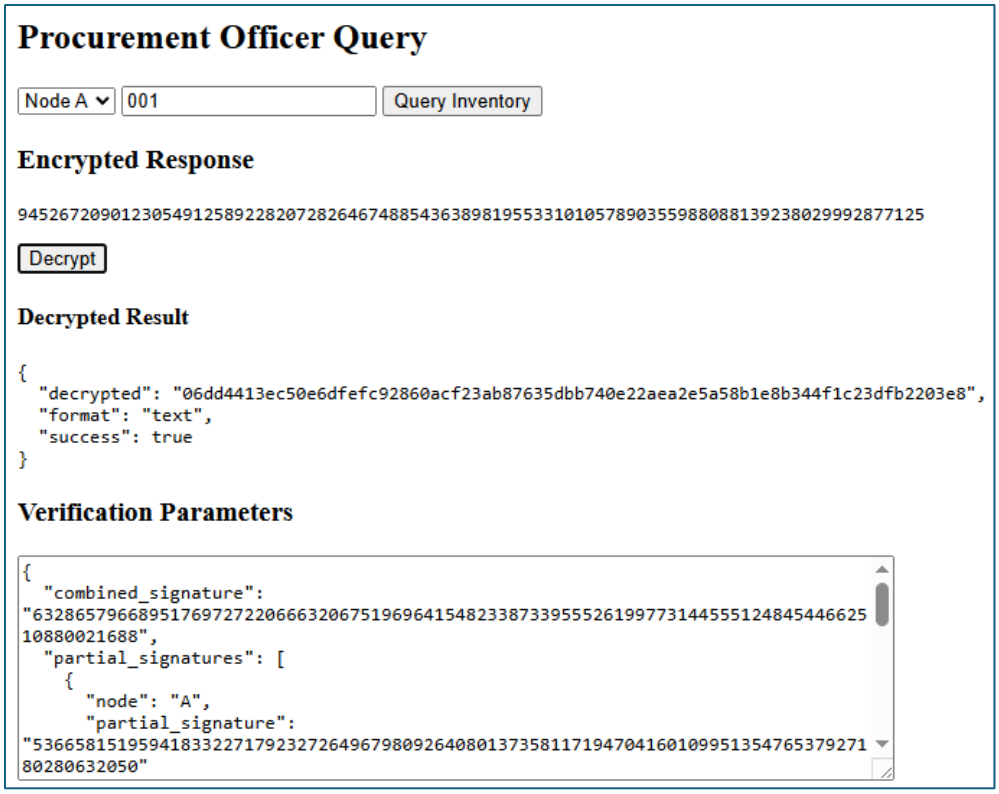
When a record is successfully submitted to the blockchain, via the submit button, it will display this message:



Procurement Officer Query:

Displays the encrypted results, with the partial signatures and complete signature.

These are all rendered dynamically using the js files and the script tags in index.html



Conclusion

The implemented system successfully demonstrates the integration of advanced cryptographic techniques and consensus protocols in a distributed inventory management context. By employing RSA digital signatures, PBFT consensus, and Harn's identity-based multi-signature scheme, the system ensures data authenticity, integrity, and confidentiality. These mechanisms collectively enhance the security and reliability of the inventory management process in a distributed environment.

References

- Investopedia (2024) *What Is Proof of Work (PoW) in Blockchain?*, Investopedia website. https://www.investopedia.com/terms/p/proof-work.asp
- Investopedia (2024) *What Does Proof-of-Stake (PoS) Mean in Crypto?,* Investopedia website. https://www.investopedia.com/terms/p/proof-work.asp
- NIST (n.d.) *Proof of stake consensus model*, NIST website. https://csrc.nist.gov/glossary/term/proof_of_stake_consensus_model
- GeeksforGeeks (2024) *Minimum number of nodes to achieve Byzantine Fault Tolerance*, GeeksforGeeks website. https://www.geeksforgeeks.org/minimum-number-of-nodes-to-achieve-byzantine-fault-tolerance/
- IACR, (n.d.) *Identity-Based Aggregate and Multi-Signature Schemes based on RSA*. IACR website. https://iacr.org/archive/pkc2010/60560484/60560484.pdf