Integrating RTL design and UVM Testbench with Hyperledger Blockchain and Machine Learning for better efficiency and optimisation.

Sougata Bhattacharjee
Samsung Semiconductor India Research (SSIR)
Bengaluru, India

***Abstract*-With increasing computational complexity within the chips and ICs, and the stringent demand to meet the time to market without compromising the quality of the chip, it is of the utmost need of the hour to bring automation and modern testbench architecture into practice. The motivation behind writing this paper is to introduce Hyperledger Fabric blockchain, integrated with both RTL and UVM-based testbenches. This integration not only facilitates the automation of regression management but also enables quick coverage closure, improved infrastructure within the server space to resolve disk space-related issues and also enables collaboration of more engineers towards a common goal in a decentralized fashion. Moreover, integrating machine learning-based applications with Blockchain helps to find out unusual failures with the help of Anomaly detection and helps to create a reusable and efficient testbench by minimizing infrastructure cost and disk space issues, so that the Test scenarios with less computation complexity (Example: not with loops, latency, wait) can run directly on Blockchain.**

I. INTRODUCTION

Moore's law suggests "the number of transistors doubles every 2 years," but as per the data, the transistor count for the chip is increasing every 18 months due to the complex nature of the chips, so that they can accommodate more features within a single Integrated Circuit. To ensure that a design or product works seamlessly without any failure or malfunction, almost 70% of the ASIC design is dedicated to verification to ensure there are no silicon bugs in the design.

As the ASIC verification process becomes increasingly complex due to the nature of chips and ICs, meeting the time-to-market goal with reduced costs using traditional methods is a challenging task. One of the solutions to this problem is to make the testbench more robust and efficient, allowing for the timely correction of bugs through continuous regression execution and coverage closure with high quality. But it has its challenges: continuous monitoring, more automated flows, better disk space, and regression management are required.

Apart from this, infrastructure management in RTL and UVM Testbench is a serious issue, as a lot of time is wasted in mismatched revision control systems while cloning the database from one revision system to another. Moreover, disk space is filling up due to the automated large regression of test scenarios, which is a matter of concern.

The Hyperledger Fabric framework [4], along with Machine Learning (ML), can help resolve all the key challenges mentioned above. It can then be integrated with UVM-based Test Benches. Hyperledger Fabric is a permissioned Blockchain with restricted users who can participate in the network with the help of a Certificate Authority (CA) and Membership Service Provider (MSP), enhancing privacy and confidentiality.

Blockchain is a growing list of records or transactions connected in a distributed peer-to-peer network.
All the blocks in the network are connected securely using the SHA256 algorithm, which makes the network tamper-proof.

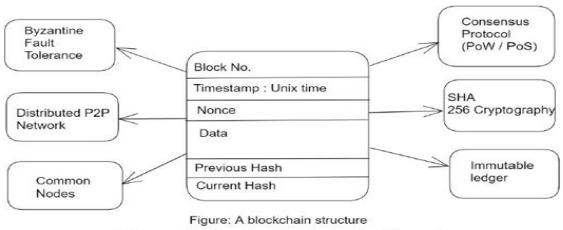


Figure: A blockchain structure

Figure 1: Basic Blockchain Structure

II. METHODOLOGY AND FLOW

a. Basic Structure of Hyperledger Fabric and Terminologies:

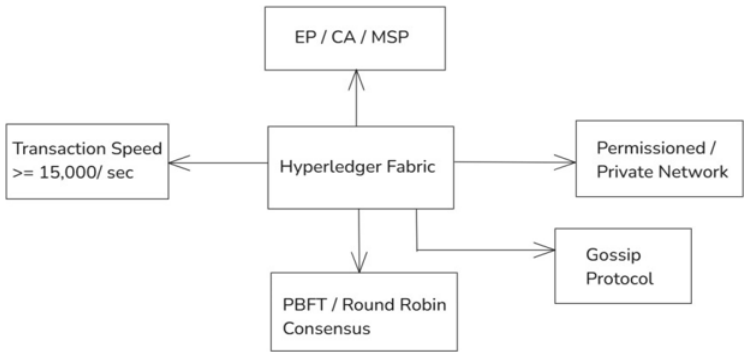


Figure 2: Basic Hyperledger components

[1] Certificate Authority (CA):

Hyperledger Fabric is a permissioned blockchain that differs from the public blockchain, like Ethereum, due to its confidential and private nature. To maintain this confidentiality, a Certificate Authority (CA) is provided to the selected peer to participate in the network.

[2] Endorsing Peer (EP):

Endorsing Peers are the members of the network who have access to vote and validate the transaction.

[3] Gossip Protocol:

The method through which transactions flow in the network. It works similarly to the spreading of rumours and is lightning fast, and the same concept is applied in this framework for the reachability of transactions.

[4] Chaincode:

Self-executing code running on the Blockchain.

[5] Practical Byzantine Fault Tolerance (PBFT) consensus:

The mechanism through which the majority of peers in the network reach a conclusion or arrive at a consensus that the transaction is valid. PBFT is faster and can validate a transaction at a speed of 15,000/sec.

b. Why Hyperledger Fabric is chosen over Public Blockchain for UVM:

[1] Provides Confidentiality and Privacy as compared to complete openness in Ethereum.

[2] No or limited Gas fees, and hence, there is no computation cost involved for coding the testbench.

[3] Transaction speed is higher (15,000/sec) as compared to Ethereum (20/sec)

[4] The PBFT consensus mechanism provides more flexibility to users as compared to the PoS (Proof of Stake) consensus in Ethereum.

c. Hyperledger Fabric framework and Integration with UVM:

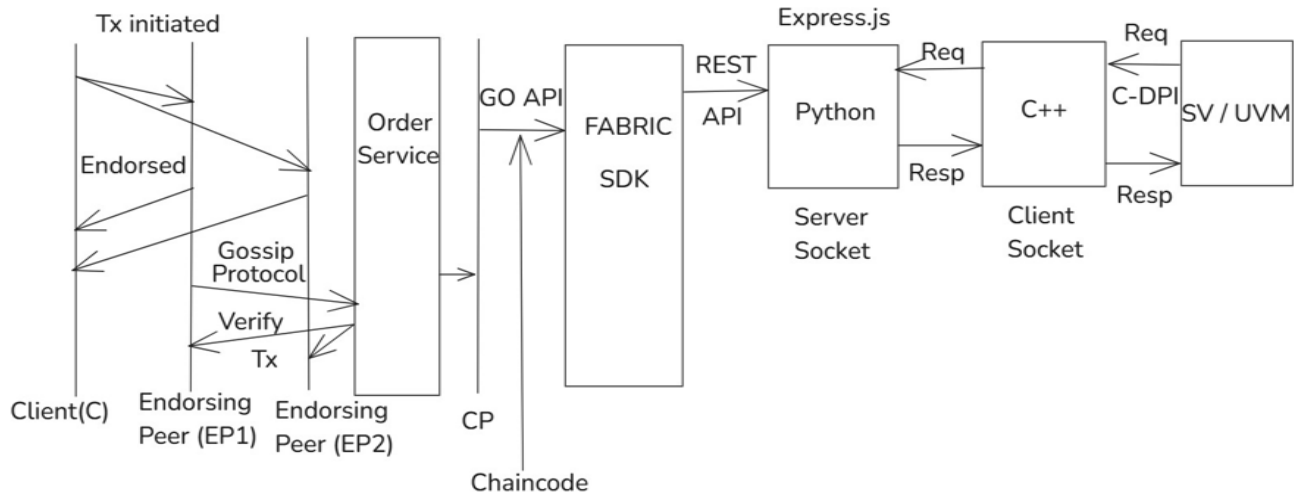


Figure 3: Integration Process of Hyperledger Fabric with UVM

[1] Hyperledger Fabric is considered for integration purposes since it is permissioned by nature and hence the data is secured; moreover, access rights can only be provided to specific individuals.

[2] Hyperledger Fabric will provide access rights depending on the Membership Service Provider (MSP). Once the network members have the access rights, they will be assigned the Certificate Authority (CA) to participate in the voting process.

[3] The network has multiple Endorsing peers (EP), which are connected in the network to validate the transactions. All the participating authorities in the network use the PBFT Consensus mechanism to arrive at a decision to add the blocks to the network. PBFT Consensus has a faster execution speed in comparison with traditional consensus mechanisms like PoW and PoS.

[4] The protocol that is followed by the EP to process the information to the ordering service is the Gossip protocol. The gossip protocol acts similarly to rumours, and more rumours are given to a specific block; that block is added to the network.

[5] Once the message is approved by the ordering service with the help of a smart contract, which is a self-executing code running on the blockchain, it will move forward to commit to the channel. In the case of Hyperledger Fabric, smart contracts are called chaincode[4].

[6] Once the channel is committed, the GO API is implemented for that specific IP for which the testbench is going to be implemented in UVM.

[7] After the Go API is integrated with the fabric code, a client-server socket connection is established with Python, which then interacts with SV and UVM Testbenches with C-DPI interfaces, as mentioned in the above diagram

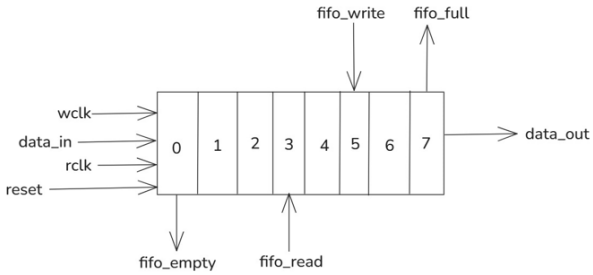


Figure 4: An asynchronous FIFO

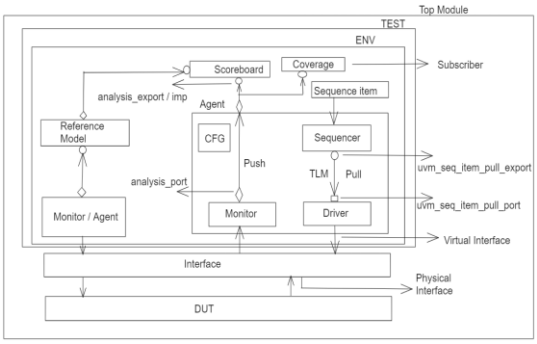


Figure 5: UVM Infrastructure

d. Integration of Fabric with Python:

One of the most important building blocks in this paper is the integration of Fabric SDK with Python, enabling the creation of a client-server and socket connection, which later facilitates integration with UVM-based testbenches. For that, DPI calls need to be made that integrate C with UVM.

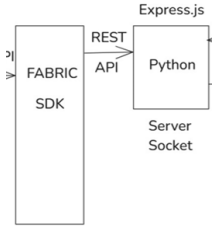


Figure 6: Integration of Fabric SDK with Python (same as Figure 3)

As can be seen from Figure 3, the GO API is connected to the Fabric SDK first, and then the whole process starts as mentioned below. But before going forward with the integration steps, it's important to know the role of the GO API.

Role of GO API:

GO API is the core foundation layer of Hyperledger Fabric, which is useful for Chaicode (A self executable code that runs on Blockchain and can not be modified) execution, transaction validation, peer review and a key parameter in connecting Blockchain with Semiconductor/VLSI. In the context of the paper where FIFO is taken as a reference, the GO API acts as a Smart contract on top of FIFO RTL (Basically A wrapper of self-executable code written in the GO language, which defines the rules of FIFO or in short working principle)

Integration Steps:

[1] The fabric sample code hierarchy consists of fabric_client.js and python_server.py, which help connect the bridge between the fabric and Python.

[2] But that is not straightforward, as corresponding users need to be registered, and the admin needs to be enrolled, and for that, corresponding codes need to be executed.

[3] Once that is done, an organisation code in JSON format needs to be written, which includes the peer details as well as the corresponding certificates, which need to be validated by the admin and the registered users.
Once these three processes are done, the connection is established.
So in short,

(fabric_server.js, python_client.py, organization.json, admin.js and regusers.js) codes are written and executed.

e. Integration of Python Server ← → C Client ← → C-DPI ←→ SV/UVM

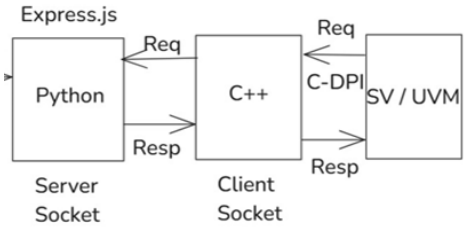


Figure 7: Connecting Python to UVM

[4] The next step is to connect the Python and C code with the request and response. After that, DPI ←→ C calls are made, which are connected with the UVM Testbench [3]

[5] Here, for reference purposes, an Asynchronous FIFO testbench is designed, which is integrated with a Blockchain environment for better coverage and regression management, easy infrastructure and more thorough verification as a lot of engineers can contribute at the same time (also discussed elaborately in Results)

Summary:

In short, Hyperledger Fabric Blockchain connects with Fabric SDK through the Go API, and then it connects with SV/UVM Testbench specific to an RTL design with the help of Python Server and C/C++ Client Socket connection.

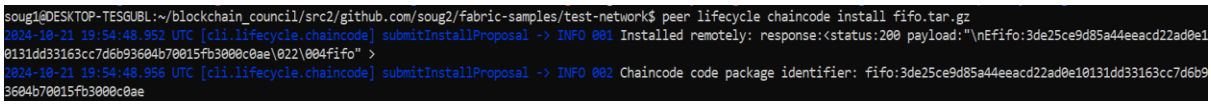
f. Compilation Steps (In brief – Execution steps):

a. ./network.sh up createChannel → Start the network
b. go build fifo.go → Compile the GO API for FIFO (GO Implementation of Async FIFO RTL – A Smart Contract of FIFO)
   ** build is clean with 0 errors
c. peer lifecycle chaincode package fifo.tar.gz --path ../asset-transfer-basic/chaincode-go/fifo --lang golang --label fifo

```
soug1@DESKTOP-TESGUBL:~/blockchain_council/src2/github.com/soug2/fabric-samples/test-network$ peer lifecycle chaincode package fifo.tar.gz --path ../asset-transfer-basic/ch
aincode-go/fifo/ --lang golang --label fifo
soug1@DESKTOP-TESGUBL:~/blockchain_council/src2/github.com/soug2/fabric-samples/test-network$ ls
README.md  addOrg3  basic.tar.gz  channel-artifacts  configtx  docker  fifo.tar.gz  log.txt  memory.tar.gz  network.sh  organizations  scripts  system-genesis-block
```

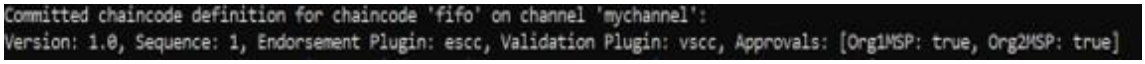
** fifo.tar.gz is created

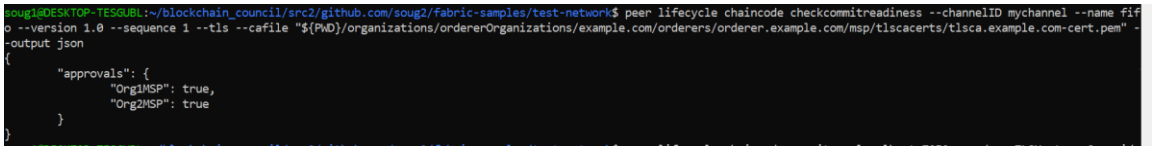
d. peer lifecycle chaincode install fifo.tar.gz →Basic chaincode command of Hyperledger



e. peer lifecycle chaincode queryinstalled

f. peer lifecycle chaincode querycommitted --channelID mychannel --name basic --cafile "$*{PWD}*/organizations/ordererOrganizations/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem"





The chaincode invoke command and orderer TLS for Hyperledger installation, which will further interact with Fabric SDK and then with UVM Testbench, is shown below:

```
peer chaincode invoke \
  -C mychannel -n basic \
  -c '{"Args":["CreateAsset","asset1","blue","10","Tom","100"]}' \
  --peerAddresses peer0.org1.example.com:7051 \
  --peerAddresses peer0.org2.example.com:9051 \
  --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt \
  --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt \
  -o orderer.example.com:7050 \
  --ordererTLSHostnameOverride orderer.example.com \
  --tls \
  --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/tls/ca.crt \
  --waitForEvent
```

g. Code Snippet and Snapshot of the framework (Fabric client – Python Server – C Client – SV/UVM):

```python
import socket
import json

HOST = "127.0.0.1"
PORT = 6000

print("[Python] Starting server...")

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((HOST, PORT))
server.listen(1)

print(f"[Python] Listening on {HOST}:{PORT}")

while True:
    conn, addr = server.accept()
    print("[Python] Connected from", addr)

    data = conn.recv(4096).decode()
    if not data:
        conn.close()
        continue

    print("[Python] Received:", data)

    request = json.loads(data)

    #this is where Hyperledger invoke/query will go later
    response = {
        "status": "SUCCESS",
        "tx": request
    }

    conn.sendall(json.dumps(response).encode())
    conn.close()
```

Figure8: fabric_client.js code

```javascript
'use strict';

const net = require('net');

const HOST = '127.0.0.1';
const PORT = 6000;

console.log("Fabric connecting to Python server...");

const client = new net.Socket();

client.connect(PORT, HOST, () => {
    console.log("Connected to Python server");

    const request = {
        command: "query",
        args: ["ReadAsset", "asset1"]
    };

    client.write(JSON.stringify(request));
});

client.on('data', (data) => {
    console.log("Response from Python:", data.toString());
    client.end();
});

client.on('close', () => {
    console.log("Connection closed");
});

client.on('error', (err) => {
    console.error("× Connection error:", err.message);
});
```

Figure9: python_server.py code.

The execution sequence:

[1] python3 python_server.py

```
soug1@DESKTOP-TESGUBL:~/blockchain_council/src2/github.com/soug2/fabric-samples/test-network$ python3 python_server.py
[Python] Starting server...
[Python] Listening on 127.0.0.1:6000
```

[2] node fabric_client.js

```
soug1@DESKTOP-TESGUBL:~/blockchain_council/src2/github.com/soug2/fabric-samples/test-network$ node fabric_client.js
Fabric connecting to Python server...
Connected to Python server
Response from Python: {"status": "SUCCESS", "tx": {"command": "query", "args": ["ReadAsset", "asset1"]}}
Connection closed
```

[3] g++ -fPIC -shared fabric_client.cpp -o libdpi_client.so

libdpi_client.so is generated, which is used in the C-DPI interface to connect with SV/UVM Testbench by using the import DPI-C function.

h. Preference of Hyperledger over CI/CD framework:

Jenkins is a CI/CD framework used to automate pipelines and build or test frameworks across multiple Developers within a centralised space and a single trust domain. The primary objective of this research work is to establish a decentralised environment rather than a centralised space to utilise the Hyperledger server for compiling and executing test scenarios with low computational complexity. Additionally, it aims to maintain an audit trail and an immutable record, which is not possible through Jenkins.

Although initially Jenkin has been considered for lower latency, faster outcomes and storing CI artifacts but, the trade-off with higher operational overhead in Hyperledger and the other advantages mentioned above make the permissioned blockchain a better choice to handle the complex UVM Testbench in a decentralised fashion. One more advantage of Hyperledger, it stores the metadata/hashes, but in the case of Jenkins, the disk might grow unbounded.

i. How Machine Learning Can Help in an Integrated Blockchain Network:

In the Integrated Hyperledger Blockchain, the Anomaly detection feature of Unsupervised Machine learning has been implemented, which helps categorize UVM failures and then bucketize them based on the error signature.

In UVM, the bugs can be categorized on the basis of their failure signature, but there are outliers as well, which will not fit into this category, and for these features, like anomaly detection, is useful. This Anamoly detection based ML learning mode helps to find out the protocol violations and corner case scenarios. Hence, for this paper Integrated Framework of Hyperledger Framework + Unsupervised learning ML model is implemented.

This type of model is specifically useful for automatic capture of failure modes and error signatures without prior labelling. The Integrated Hyperledger further helps in coverage tracking, regression management, bug fix reports, automates the verification flow, and most importantly, enables the low computation test scenarios to use the blockchain server rather than the central server to minimize disk space usage.

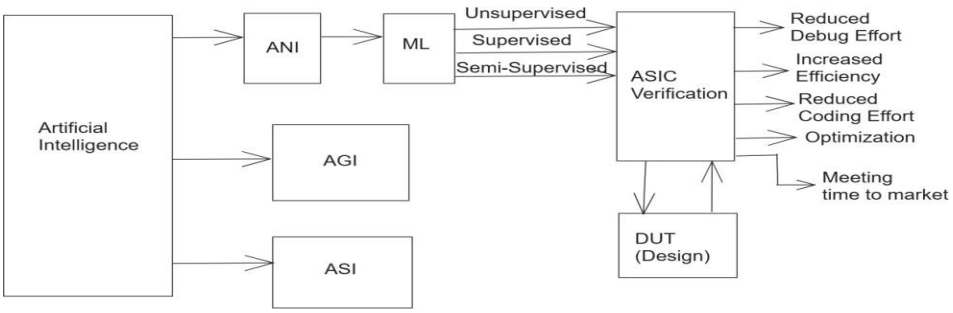


Figure 10: Integrated Machine learning applied to Hyperledger

III. COMPARISON OF ETHEREUM WITH HYPERLEDGER

An alternative solution has been tried to integrate the UVM testbench / RTL with the Ethereum blockchain, as shown in Figure 10. A smart contract, which is a self-executing code running on blockchain and is non-modifiable by nature and is written in Solidity coding language [1][2], has two main parts: Application Binary Interface (ABI) and a Bytecode. The ABI is particularly used to connect with the outside world, and it's possible only with the help of web3.js or ether.js.

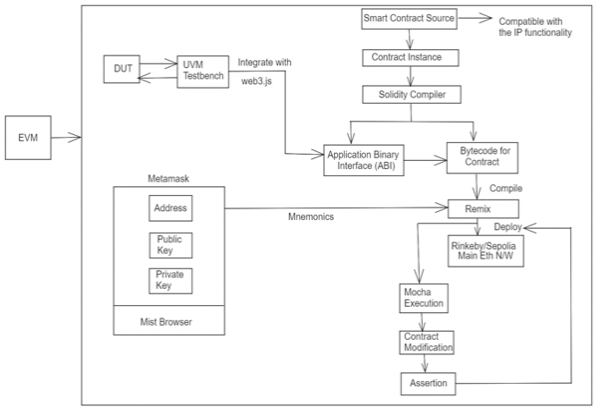


Figure 11: Framework of integrating Solidity with UVM and RTL

But the main problem as to why Ethereum is not a preferred solution is that it is a public Blockchain, and the project execution, which took place in a company, requires confidentiality and privacy. Hence, due to that, Hyperledger is considered. Moreover, as computational complexity increases, GAS Fees become a problem in Ethereum (A fee needs to be paid to execute the code in the Blockchain), which is not the case with Hyperledger, as its transaction speed is quite higher (~15,000/sec) than Ethereum. Due to these reasons, Hyperledger is considered a more innovative and desired approach.

IV. Results:

- The infrastructure cost of the server has been reduced by ~25% due to blockchain technology, as it helps to alleviate disk space issues substantially that arise from massive regression *(calculated based on how many Test scenarios can utilize the Hyperledger blockchain rather than central disk space)*

- The interdependency in the revision control system will be reduced and save time for both the verification and RTL Engineers, resulting in unnecessary *debugging (often the case when the RTL and Verification Engineers have separate databases and use different revision control systems).*

- The security issues with the hardware chips have been significantly reduced by 20% to 30% since Blockchains are cryptographically linked with each other, making the system tamper-proof and restricting any unauthorized access. IP protection has also become easier due to this framework.

- Meeting the time-to-market and executing complex projects will be faster as more engineers can collaborate on a private, decentralized server to work together towards their completion.

- Further implementing Anomaly detection using unsupervised learning in the UVM testbench helps categorize errors based on their signature and also detects any exceptions to them.

- With the implementation of Smart Contract / Chaincode, bug tracking will be easier as the bugs can be easily filtered and segregated depending on their signatures, as smart contracts are immutable.

- The permissioned Blockchain, like Hyperledger, always maintain the confidentiality and Privacy with the help of a Certificate Authority and at the same time makes the system tamper-proof and extremely secure.

V. Future Scope and Challenges:

Hyperledger Fabric uses an ECDSA cryptography algorithm for signing transactions and an AES symmetric encryption mechanism for data privacy. Moreover, today's Machine learning algorithm, which is applied to perform a certain task on Blockchain, also offers security. However, the challenge is that with the recent rise of Quantum computers, and as Shor's algorithm suggests, it can solve complex logarithmic and discrete factorization problems in polynomial time as compared to classical computers, making the AES encryption and other security measures vulnerable.

As a result, research is being conducted on post-quantum cryptography algorithms to make the existing Blockchain network and AI models resistant to Quantum attacks. Some of the algorithms are:

[1] Lattice Cryptography

[2] Code Cryptography

[3] Isogeny Cryptography

REFERENCES

[1] "Automatic Generation of Solidity Test for Blockchain Smart Contract using Many Objective Search and Dimensionality Reduction" – IEEE paper by Doncheng Li

[2] "How are Solidity Smart Contracts tested in open-source projects?" – IEEE Paper by Luisa Palechor

[3] "UVM User Guide" by Accellera

[4] "Hyperledger Fabric Blockchain: Chaincode Performance Analysis" – IEEE paper by Luca Foschini