Final Project Report: Mini Blockchain System Implementation

Introduction and Contributions

This report aims to design, implement, and simulate a mini blockchain system built from scratch using the Python programming language. Like a blockchain, the system can generate cryptographic transactions, develop and validate Merkle trees, form and verify blocks via proof of work (PoW) mining, and validate complete chain integrity. Our implementation is intended to provide an approach that presents blockchain mechanisms in a simplified fashion, is educational, and is reproducible, transparent, and modular.

We were a project group with three members: Ahmed Ahtasham (22229817), Chen Boyu (22240985), and GARG Shriya (24512028). There was no favoring of any member along four major components of the project: coding (Sections 4.1–4.5), reproducibility artifacts, presentation video creation, and report compilation. Ahmed Ahtasham was primarily responsible for implementing the blockchain architecture and block verification protocols and contributing appreciably to the documentation and report writing. Using RSA-based public-key cryptography, Chen Boyu designed the transaction generation system and implemented the Merkle tree to verify transactions. In addition, he also made significant simulation output contributions and co-produced the project presentation. The proof of work mining mechanism proof of work mining mechanism was developed by GARG Shriya, who also built the core blockchain and block structures. Besides that, she led the creation of the README, reproducibility documentation, and the final report as a co-author. The benefit of evenly distributed collaboration was that all the members were deeply involved in the technical and presentation sides of the project.

Related Tools and Libraries

In the project, we used Python 3.11 as it provides a straight syntax for educational demonstration, and there is enough support for cryptographic operations. The cryptography library was used to generate and verify RSA key pair, data signing, and

signature verification under PKCS#1 v1.5 using SHA-256 hashing. We used the Hashlib library from Python built inside that already implements the SHA 256 algorithm for all the hashing operations (transaction IDs, Merkle nodes, block hashes). I generated the timestamp data using the datetime and time libraries and measured the miner's times during the PoW process. The readability and code correctness of the whole implementation were improved by using the typing module for type annotations.

Pipeline Flow and Architecture Overview

The system is a linear, modular pipeline starting from user account creation. Each account has a 2048-bit RSA key pair; the public key that has the hash of SHA-256 becomes the user's blockchain address. The transactions are the model – single input and single output (SISO) transaction, which includes the sender and receiver addresses, the amount to transfer, a timestamp, and a signature. The signed content is generated using the sender's RSA private key and then hashed to form the transaction ID (TID), the remainder covered by the signature. This allows authentication as well as nonrepudiation.

After that, transactions are sent into a binary Merkle tree, which is built from the bottom and onwards by alternatively pairing and hashing a transaction ID until a single hash root is computed. It also summarizes all transactions of the block through a cryptographic Merkle root. Finally, the Merkle root, combined with the previous block hash, timestamp, and a nonce value, is added to the block. The problem of finding a hash less than or equal to the target is used as a proof-of-work for this mining process, thereby allowing a running chain to remain secure from concurrent chains, reflected by the assumption of a large number of intermediary chaining steps. A block is successfully mined, and the previous block hash pointers are linked to the block and updated on the blockchain.

Finally, transactions and blocks are checked for integrity and verified as authentic. With a sender's public key, digital signatures can be verified, and Merkle tree proofs can be used to validate that the included transaction is part of any transaction. In any recomputation or comparison, the chain's immutability is preserved by detecting

changes in transaction data, timestamps, or hash linkages.

Design and Implementation Details

In the implementation, there's an Account object for which each account generates/creates and stores an RSA key pair. Then, the public key is serialized in PEM format, and its content is hashed to get a unique blockchain address. The account's private key is used to sign the transaction data of sender and receiver addresses, transfer amount, and timestamp. Combining the transaction data and signature, using SHA-256 to hash them, gives the transaction ID (TID).

Under the assumption that the number of transactions is a power of two, one starts with the leaf nodes, which are the hash of TIDs and constitute the Merkle tree. The concatenation of the hexadecimal values of child nodes and hashing them create internal nodes. A recursive process will generate a root hash, which can be used to verify transaction inclusion and is trusted against data tampering. Merkle paths are also proof of inclusion.

It works on the proof-of-work principle inspired by Bitcoin's mining protocol. It tries to find a nonce that produces a block hash with a certain number of leading zeros. It iteratively recalculates the block header hash (commonly referred to as the hash), which is composed of the previous block hash, content hash, timestamp, and nonce. It sets its result to the difficulty threshold when the result equals the target.

Genesis block is created with a pseudo transaction that moves from GENESIS to NETWORK (seeding the blockchain). After the latest block, a reference is made to its hash, and subsequent blocks are built using the previous block's hash. Verification includes the integrity check that the stored hash matches the recomputed one, that the Merkle root is the same as the transaction set, and that the previous hash link isn't being explained inappropriately.

We also enabled specific modules to simulate attacks by modifying transaction content, block timestamps, and previous hashes. Our system is resilient against each tampering attempt by being caught on hash mismatches or Merkle root inconsistencies.

Experimental Results Based on Simulation

```
=== Building Merkle Tree ===
Leaf nodes:
Leaf 0: fe97db694d3425a71a968ec04288465075e7e1d04365fd22bf6
62879b2ee59df
Leaf 1: 1d9d358e96f9285943cb95eea92053e3508a339430b783937da
c8687a59d8214
Leaf 2: e96a6dc1d11bb5b3461bebcf50e4bd0b71931d3e7ace81e4f72
1c6914edcaf5f
Leaf 3: 5e3bfd6acc20915f4a77f076925976471a7c9325d9dc0e016df
3088215b6bcb8

Level 0:
Combining fe97db69... and 1d9d358e... -> 754919ce...
Combining e96a6dc1... and 5e3bfd6a... -> 0e2ac0f4...

Level 1:
Combining 754919ce... and 0e2ac0f4... -> 1aed2219...

Root hash: 1aed22193d528c512b61564a603423cb5d39da8a3e22ae11
fdd1aaed9470019b
Mining block with difficulty 4...
Block mined! Nonce: 2361, Hash: 00003a3b36463527250aaa04040
311127beebcb0984df8e225310bd6e3db4252
```

```
=== Verifying Blockchain Integrity ===
Blockchain is valid!
Chain valid: True
```

```
=== Simulating Blockchain Attacks ===

1. Attempting to modify transaction amount...
Modified transaction amount from 100 to 200
```

```
2. Attempting to modify block timestamp...
Modified block timestamp
```

```
3. Attempting to modify previous hash...
Modified previous hash
```

For instance, simulations were carried out in which two blocks were used, each with four transactions, and proof of work mining was done at a difficulty requiring four leading zeros in block hash. Finally, the mining process was probabilistic in that the

nonce values needed to mine blocks varied greatly. We tested real-world variance in that one block takes about 2,300 iterations to mine while the other gets more than 90,000 iterations.

We also demonstrated the correctness of the Merkle root construction with our implementation. Since the Merkle tree structure of the Merkle root is sensitive and reliable, as long as a single character in a transaction could change the corresponding Merkle root, the block would be invalidated, and the transaction would be changed. All valid entries were successfully verified to a transaction through public-key signatures and failed appropriately on any forged or modified data.

The system was able to detect alterations in the attack simulation module. A mismatched hash resulted in a failed integrity check when changing a transaction amount or timestamp. Like this, modifying a block's hash from the previous one broke the blockchain verification function's bond, meaning that the overall chain structure was invalidated. Chaining hashing and cryptographic verification in blockchain systems is essential, as revealed by these results.

Conclusion

In this project, a fully working, modular mini-blockchain system is made in Python, implementing everything most blockchain platforms offer today in a minimalistic but robust shape. Using RSA cryptography, SHA-256 hashing, Merkle tree validation, and proof of work mining, the system presents a basic grasp of how blockchain technology vouches for trustless, immutable data, a consensus amongst nodes, and a secure transaction medium. We built everything from scratch and verified that each part of the system was resilient through simulation and various attack scenarios. Our design decisions prefer security, clarity, and modularity to assist educational exploration and reproducibility.

References

Apache Software Foundation. (n.d.). DigestUtils (Apache Commons Codec 1.15 API). https://commons.apache.org/proper/commons-

codec/apidocs/org/apache/commons/codec/digest/DigestUtils.html

Bitcoin Wiki Contributors. (n.d.). Protocol documentation: Merkle Trees. Bitcoin Wiki. https://en.bitcoin.it/wiki/Protocol_documentation#Merkle_Trees

Dutta, A. (2022, June 6). Building a blockchain in Python. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2022/06/building-a-blockchain-in-python/

PyCryptodome Project. (n.d.). RSA encryption and decryption. PyCryptodome Documentation. https://pycryptodome.readthedocs.io/en/latest/src/public_key/rsa.html

Python Cryptographic Authority. (n.d.). RSA — cryptography 42.0.5 documentation. Cryptography.io. https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/

Python Cryptographic Authority. (n.d.). Serialization — cryptography 42.0.5 documentation. Cryptography.io. https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/

Python Software Foundation. (n.d.). cryptography 42.0.5. PyPI. https://pypi.org/project/cryptography/

Python Software Foundation. (n.d.). datetime — Basic date and time types. Python documentation. https://docs.python.org/3/library/datetime.html

Python Software Foundation. (n.d.). hashlib — Secure hashes and message digests. Python documentation. https://docs.python.org/3/library/hashlib.html

Python Software Foundation. (n.d.). typing — Support for type hints. Python documentation. https://docs.python.org/3/library/typing.html

The Update Framework. (2022). securesystemslib 0.14.2. PyPI. https://pypi.org/project/securesystemslib/0.14.2/