CISC 322/326 Assignment 2: Report
Bitcoin Core: Concrete Architecture Analysis
March 15, 2023

Group 6: ArcProject
Chuyang Li (19cl98@queensu.ca)
Qintao Zhang (18qz28@queensu.ca)
Anthony Zhou (18zz172@queensu.ca)
Chang Xu (18cx19@queensu.ca)
Wenchu Xiao (19wx25@queensu.ca)
Xiaoran Zhang(19xz64@queensu.ca)

Table of Content

1. Abstract

After an in-depth examination of the Bitcoin Core source files and summarizing its concrete structure, we found divergences between the concrete and conceptual structures, which allowed us to identify differences between the theoretical and practical implementation of the system. Therefore, it is clear that we believe that refining the conceptual architecture is necessary. In this report, we present a more precise and understandable representation of the Bitcoin Core architecture. Our analysis of Bitcoin Core will rely on the refined conceptual framework as a foundation while using the source code to confirm our findings.

2. Introduction and Overview

The purpose of this analysis is to give a more comprehensive view of the concrete architecture of Bitcoin Core. We used the Bitcoin Core documentation and its source code to derive the concrete architecture.

We decompose the subsystems in the architecture and make reasonable speculations about their relationships and how they interact with each other. Although our previous conceptual architecture provided us with the main framework to build our article, after our deep analysis, we found many dependencies that were not in our last conceptual architecture. So this article will first explain the adapted conceptual architecture, and then discuss in depth the concrete architecture we have summarized.

The analysis will mention top-level concrete subsystems and their interaction with notes, propose and introduce new subsystems and include use cases. moreover, we will further explore the internal architecture of the Wallet component in detail with a conceptual and concrete view. Finally, we will summarize the analysis and will conclude this analysis with a summary of our findings and a proposal for the rest of the assignment.

3. Derivation Process

At the beginning of this process, we concluded that the architectural style of Bitcoin Core is P2P by reviewing the feedback from Assignment 1 and the report from other groups. It is also said in the developer guide of Bitcoin Core “The Bitcoin network protocol allows full nodes (peers) to collaboratively maintain a peer-to-peer network for block and transaction exchange.”[1] According to suggestions from other groups, the previous conceptual architecture of Bitcoin was analyzed for shortcomings and planned to be optimized in concrete architecture.

In the second phase, we reviewed the Bitcoin Core source code and reduced the main components to 10 generic components. For a deeper analysis, we studied all the files in the src directory and grouped each file into components with similar functionality based on their function in the project. Next, we generated dependency graphs and discovered some components and their dependencies that were not mentioned in the previous conceptual

architecture. Based on this, we analyzed the logical relationships between the new dependency graph components and the previous conceptual architecture components in the Bitcoin project to map out the specific architecture.

In the third phase, we performed a detailed comparative analysis of the final concrete and conceptual architecture diagrams, going through the differences between the two diagrams and delving into the root causes of any unexpected dependencies.

4. Conceptual and Concrete Architecture

Bitcoin Core adopts a modular design approach, with individual components communicating with each other through clearly defined interfaces, enabling an effective separation of concerns. Such a design facilitates efficient development and maintenance of the cryptocurrency platform, allowing individual components to be updated independently.

After extensive research and analysis of Bitcoin Core, we have obtained Figure 1, which shows the final concrete architecture of the Bitcoin Core software system. This diagram integrates dependency diagrams and event-driven communication diagrams to clearly show the connections between the various modules.

The underlying framework for the concrete architecture diagram is derived from the conceptual architecture diagram, which depicts key components such as the user interface, programming interface with RPC, consensus mechanism, encryption, memory pool, mining, peer-to-peer network, storage, authentication, and wallet. To more clearly show the complex relationships between these modules, we have included red dashed lines in the diagram to highlight some unanticipated dependencies and newly discovered subsystem modules.

Here are the reasons that both architectural views are different:

1. Some subsystems are essential in program execution but are not mentioned in the general architecture in the developer guide or open book. Such that Cryptographic and Consensus Algorithms.
2. There are some components that are not dependent on other components like the conceptual architecture, when we looked into the source code, we found that some of our assumptions are not applicable. That the dependency from mining to wallet and mempool is gone due to there being no dependency at the source code level.
3. There are also lots of dependencies we were not able to identify when we analyze the general architecture. After reviewing the source code, we found that there are more dependencies than we thought between those components. Although they are independent subsystems, they all need to interact in order to operate successfully.
4. Some components in our previous work should be combined with other components since it’s lower level compared to others. Such as transaction is combined into the wallet component.

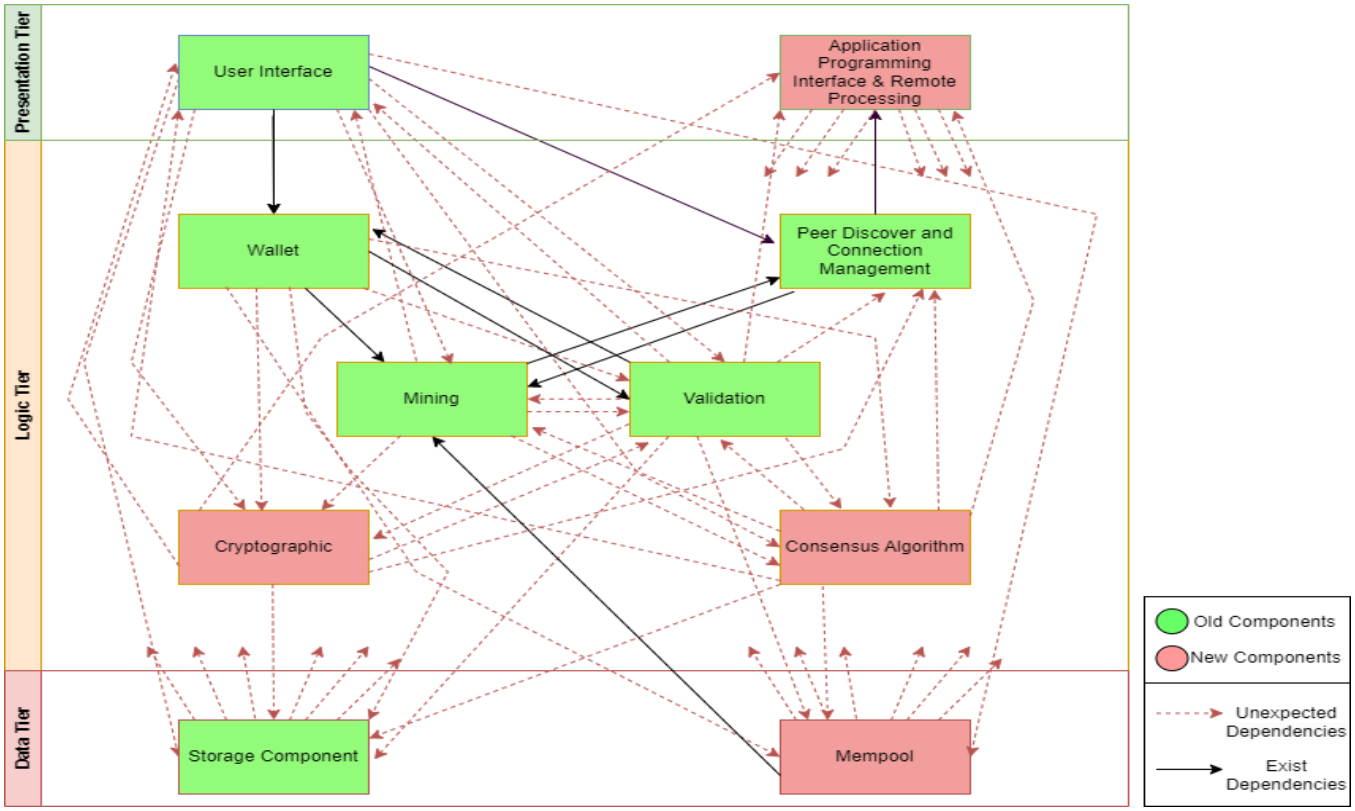

Figure 1. Top-Level Concrete Architecture

4.1 Subsystem Overview

Transaction*:* Responsible for processing, validating, and broadcasting transactions to the network, ensuring their authenticity and compliance with network consensus rules before adding them to the blockchain through the mining subsystem.

Validation(Script): Responsible for executing transaction scripts to verify their validity and ensure they are authorized by the bitcoin owner, interpreting instructions that establish conditions for spending the bitcoin being transferred. If the script passes validation, the transaction moves to the next stage, but if it is deemed invalid, the transaction is rejected.

Wallet: This component's main function is to facilitate the sending and receiving of Bitcoin transactions and store the private keys required to sign and verify those transactions and keep an accurate record of a user's transaction history and balance and can generate new addresses for receiving funds.

Mining: This component is responsible for the creation and validation of new transactions on the blockchain and for generating new blocks containing transaction data, which are then added to the existing blockchain while verifying the validity of the transactions.

Block Chain:

This component is responsible for the core of the Bitcoin network. This structure ensures that all transactions on the blockchain are linked together in a tamper-evident manner, providing a transparent, immutable and secure record of all transactions on the network.

Peer Discover and Connection Management(Network):
The network subsystem in Bitcoin Core is responsible for establishing and maintaining connections to other nodes on the Bitcoin network. It also facilitates the propagation of new transactions and blocks throughout the network.

4.2 New Subsystem
Application Programming Interface & RPC:
API provides a simplified, high-level interface for developers to interact with the software, and the RPC interface allows for direct access to the software's functionality, providing greater control and customization options. API is considered to be more user-friendly and easier to use, whereas the RPC interface offers more direct and granular control over the Bitcoin Core.

Wallet:
Provides functionality for creating and managing Bitcoin wallets. It helps with key generation, transaction signing, and backup and recovery.(Transaction subsystem now is included in the Wallet subsystem.)

Validation:
Responsible for verifying the validity of transactions and blocks. Ensures that transactions have correct digital signatures, are properly formatted, and comply with the rules of the Bitcoin protocol. Checks that blocks are constructed correctly and correctly link to the previous block in the blockchain.

Consensus Algorithm:
Defines the rules for validating transactions and adding new blocks to the **blockchain**, including the proof-of-work algorithm used for mining and the difficulty adjustment algorithm for regulating block creation.

Mempool:
Organizes a pool of unconfirmed transactions waiting to be included in the next block and includes rules for prioritizing transactions, preventing double-spending, and managing transaction fees.

Peer Discover and Connection Management: Enables modes in the Bitcoin Core to communicate with each other in a decentralized and Peer to Peer mode. Includes protocols for establishing connections, exchanging messages, and managing peer relationships.

Storage: Responsible for securely storing the entire blockchain and other crucial data related to the network's operation.

Cryptographic libraries:
This component supports the **cryptographic primitives** necessary for the security of the Bitcoin network. It includes features such as algorithms for hashing, **digital signatures**, and **encryption**.

5. Perception (Secondary Subsystem Conceptual and Concrete View)

5.1 Conceptual View

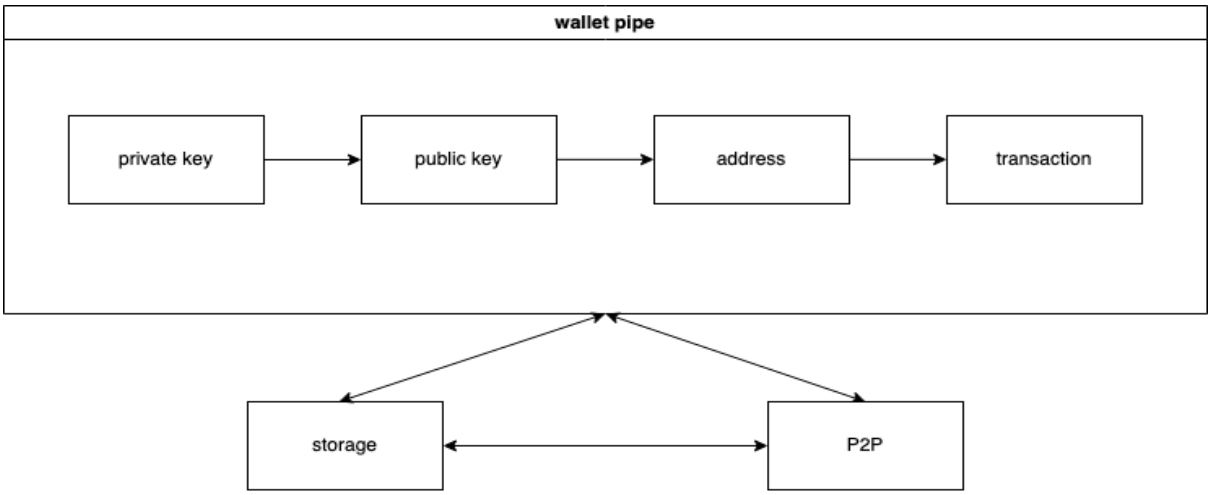

Figure 2. Conceptual Architecture of Wallet

5.1.1 Inner Components Introduction

Private key: The secret number is used to sign transactions that transfer ownership of Bitcoins. It is critical to keep the private key secret, as anyone who has access to it can transfer the Bitcoins associated with that key.
Public key: This component is responsible for deriving the public key from the private key through a mathematical process called Elliptic Curve Cryptography (ECC).
Address: This component is generated from the public key using a series of hash functions for ensuring that the address is unique and cannot be guessed.
Transaction: When someone wants to send Bitcoins to a specific address, they create a transaction that references the UTXOs associated with that address. The transaction includes the address of the recipient and the amount of Bitcoins being sent.
Storage: This component is critical to maintaining the security of Bitcoin: The storage of private keys is critical to the security of Bitcoin. Private keys must be kept safe and secure to prevent unauthorized access to a user's Bitcoin.
P2P: This component is responsible for the transfer of Bitcoin as transactions are broadcast to the network and validated by nodes on the network. Without P2P communication, it would be impossible to transfer Bitcoin from one user to another.

5.2 Concrete View

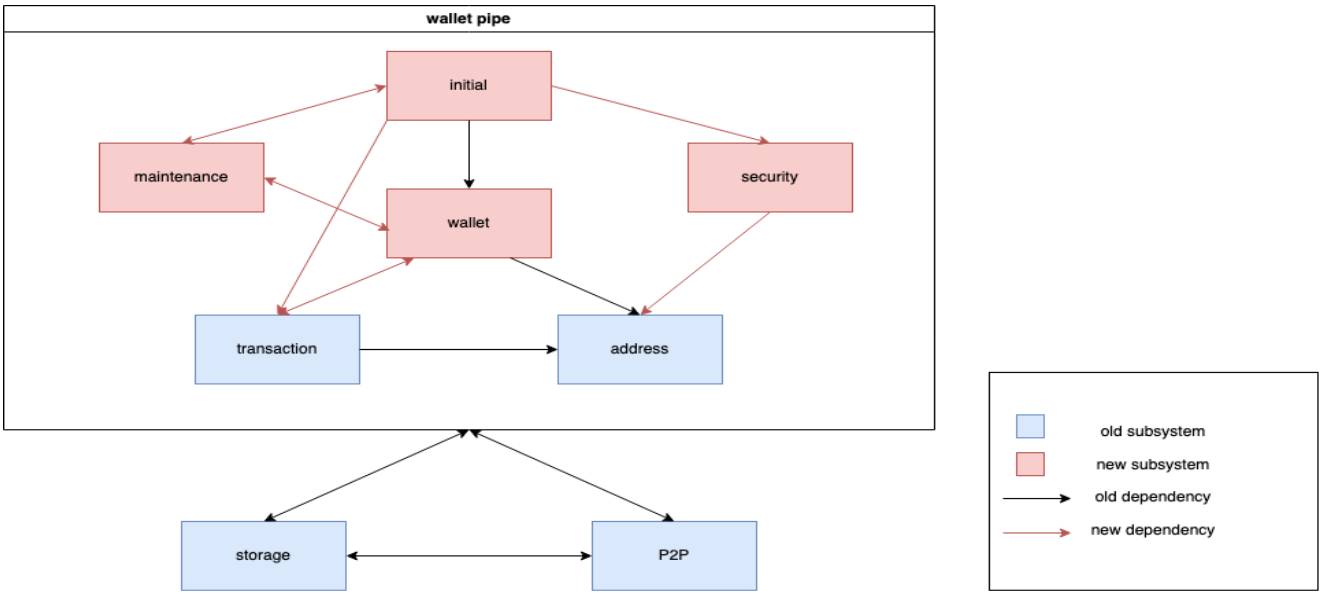

Figure 3. Concrete Architecture of Wallet

5.2.1 New components

Initial:
In Figure 3, the Initial component is responsible for initializing the Wallet subsystem and setting it up for use. The files in the Initial are used to store the private keys and transaction history.

Wallet: The Wallet component in Figure 3, the primary function is to manage the user's private and public keys, addresses, and transaction history. It is responsible for generating and storing private and public keys used to sign transactions.
Security:
The security component is critical for protecting the users' assets and ensuring the system's integrity. It protects the other components in the Wallet Subsystem from information leakage or unauthorized transactions.

Maintenance:
The Maintenance component ensures all components in the wallet subsystem remain operational and up to date. Ensuring all components are ongoing, reliable, usable, and secure is crucial.

6. Discrepancies

We generated our own dependency graph using Understand, as depicted in Figure 4, to analyze any unexpected dependencies not visible in the conceptual view.

6.1 High-Level Unexpected Dependencies

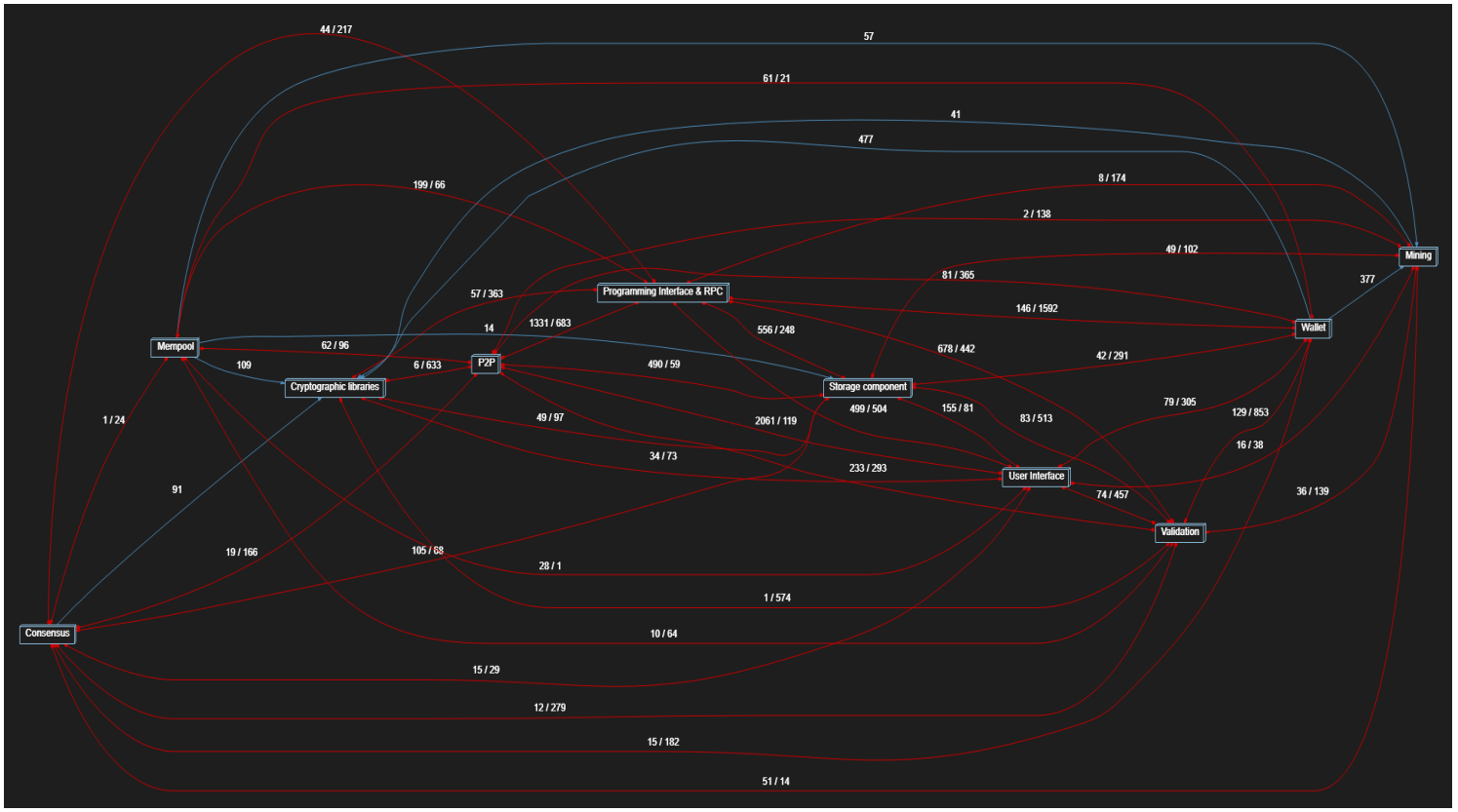

Figure 4. Dependency Diagram of Bitcoin's Architecture[2]

Consensus ↔ Programming Interface & RPC, Mempool, P2P, Storage, UI, Validation, Mining, Wallet

In Figure 4, we found that there are lots of components that are different from A1, therefore we get lots of unexpected dependencies. In the programming interface & RPC, the API follows consensus rules to ensure that the information provided to the application is accurate and consistent with the rest of the network. Transactions are validated against the consensus rules before being added to the memory pool. Nodes share and synchronize their memory pools so they know about pending transactions that follow the consensus rules. the P2P network enforces consensus rules when forwarding information, ensuring that only valid data is propagated across the network. The storage component ensures that only blocks and transactions that follow the consensus rules are added to the local copy of the blockchain. the UI communicates with other components that enforce the consensus rules, ensuring that users only interact with valid data. The validation component checks that transactions and blocks comply with the consensus rules. In the mining component, miners must follow the consensus rules when creating new blocks, ensuring that the blockchain remains consistent across the network. The wallet is responsible for creating and signing transactions, ensuring that they follow the consensus rules before they are broadcast to the network.
Rationale: A newly added component will have lots of dependencies in order for it can function properly in the system, especially when the component defines a number of rules in the algorithm.

Wallet → Mining and Mempool

Another unexpected dependency is that the mining module doesn't directly depend on the wallet and mempool modules. We initially thought that the mining module would interact with the wallet to generate new addresses for receiving rewards and with the mempool to access pending transactions. However, these dependencies were not found at the source code level.

Rationale: While certain subsystems are essential for program execution, they are not part of the general architecture outlined in the developer guide or open book.

API & RPC ↔ Wallet, Consensus Algorithm, Mempool, Storage, Validation, P2P

The API and RPC subsystems in the Bitcoin core architecture establish important connections between users, developers and components such as wallets, consensus algorithms, mempools, storage, authentication and P2P networks. The interface functions are as follows:

(1) Convenient wallet management: allows users to create addresses, send and receive transactions, query transaction history and balances, and interact with wallet components for key generation, transaction signing, backup and restore.

(2) Interaction with consensus algorithm: Get information on current network consensus rules, monitor blockchain status, and collect blockchain statistics such as mining difficulty and block height.

(3) Query Mempool: Get information about unconfirmed transactions waiting to be included in the next block, and understand the network status, such as transaction fees and confirmation times.

(4) Access to the Storage component: Query blockchain data, retrieve transaction details, get information about specific blocks, and drill down to analyze the Bitcoin network and historical data.

(5) Interact with the validation component: retrieve information about the validity of transactions and blocks, ensure compliance with network consensus rules, and verify the transaction and block correctness.

(6) Interacting with P2P network components: Querying connection peer information, establishing new connections, monitoring the overall health of the network, and gaining real-time insights into the connectivity of the Bitcoin network.

Rationale: API and RPC modules exhibit broader connectivity to other subsystems in specific architectures than initially exhibited in the top-level conceptual architecture. This difference comes from the greater detail and specificity provided by the specific architecture, which comes from the actual source code.

Validation (Script) ↔ Cryptographic Libraries

The validation module is responsible for verifying the validity of transaction scripts, ensuring their proper execution, and checking if the conditions for spending bitcoins are met. In the concrete architecture, the validation module may have an unexpected dependency on

cryptographic libraries, as it needs cryptographic primitives for tasks such as signature verification and hashing.
Rationale: The validation module needs cryptographic functions to validate transactions, which might not be evident in the conceptual architecture.

Storage ↔ Cryptographic Libraries

In the conceptual architecture, the storage subsystem may not show a direct dependency on cryptographic libraries. However, in the concrete architecture, the storage module might rely on cryptographic primitives to securely store sensitive data such as private keys, ensuring the security and integrity of stored data.
Rationale: The storage module may require cryptographic functions to protect sensitive data and maintain the integrity of the stored blockchain data.

Cryptographic Libraries → Consensus Algorithm

The Consensus Algorithm component also has an unexpected dependency on the Cryptographic Libraries. Cryptographic hashing functions, such as SHA-256 and RIPEMD-160, play a vital role in the proof-of-work algorithm used in the Bitcoin mining process. These functions are used to create a unique hash for each block, which miners then try to solve by finding a nonce that results in a hash meeting the target difficulty.
Rationale: The initial analysis did not specifically mention the role of cryptographic hashing functions in the consensus algorithm, leading to an underestimation of the dependency between these two components.

Cryptographic Libraries ↔ Peer-to-Peer (P2P) Network

In the Bitcoin core architecture, the cryptographic library is tightly integrated with the peer-to-peer (P2P) network components that work together to maintain the security and stability of the system. the P2P network handles connections and communications between nodes, and the cryptographic library provides the necessary security for these communications. The encryption function ensures the confidentiality, integrity and authenticity of the information transmitted between nodes. In addition, using the distributed network feature, encryption tasks can be distributed across the network, thus enhancing the security and resilience of the entire system against attacks. This interdependence and synergy improves the resilience and efficiency of the Bitcoin network in response to potential security threats.
Rationale: Our initial analysis did not fully explore the role of cryptographic functions in securing communication between nodes, leading to an underestimation of the dependencies between P2P network components and cryptographic libraries.

Cryptographic Libraries → Mempool

In the Bitcoin Core architecture, the cryptographic library plays an important role in the operation of the Mempool component. mempool is responsible for managing unconfirmed transactions waiting to be incorporated into new blocks, ensuring that they are valid and follow consensus rules. mempool relies on the cryptographic library to verify transaction

encryption details, such as checking digital signatures and confirming sender spend permissions. The cryptographic library also computes transaction hashes as unique identifiers, which are crucial for transaction management, prioritization, and prevention of double-spending attacks in Mempool.
Rationale: In our initial analysis, the dependencies between the cryptographic library and the Mempool components were not obvious because we did not fully explore the role of cryptographic functionality in protecting and validating transactions within Mempool.

6.2 Secondary Unexpected dependencies

Wallet ↔ Maintenance, Transaction

The Maintenance was not discussed in the conceptual analysis but is introduced in the concrete architecture graph (Figure 3). The reflexion model shows that maintenance and Wallet components depend on each other because the wallet component includes the maintenance-related functionality to control the maintenance module to remain secure and accessible in case of a system file or data loss. The maintenance module also provides the functionality to update the wallet module. Meanwhile, the Transaction module and Wallet module also depend on each other. The transactions rely on the wallet's private and public keys to create, sign, and broadcast. The transaction history functionality in the Wallet module receives the information from the transaction module.
Rationale: The dependency between maintenance, transaction, and wallet components to ensure the transaction and transaction fee remain operational and up-to-date in the whole transaction case.

Initial ↔ Maintenance

The initial module and maintenance module have a dependency on each other. The initial module is responsible for creating and initializing all modules in the wallet subsystem, such as maintenance. Maintenance is responsible for ensuring the initial module's ongoing reliability and functionality because it remains the initial module operational and up to date.
Rationale: The dependency between initial and maintenance ensures the wallet subsystem remains configured correctly over time by updating the initial setting or preferences.

Initial → Transaction, Security

The dependency between initial, transaction, and security ensures the wallet subsystem remains configured correctly over time. The transaction module is responsible for managing the user's transactions, and it relies on the initial component to ensure that the wallet is properly initialized with the correct set of private keys and addresses. Meanwhile, the security component is responsible for protecting private keys. The initial module generates the initial set of the private key. Hence they have a dependent relationship.
Rationale: The initial module generates the initial set of private keys and addresses. The security components protect the private keys, and the transaction component uses private keys and addresses to create, sign, and broadcast transactions.

Security → Address

There is a strong dependency between the security and address components. The address component is responsible for generating the public addresses. These addresses are derived from private keys. The security component is responsible for protecting the wallet's private keys. Hence, the security module protects the security of the wallet's addresses. Rationale: The addresses are derived from the private key, which is protected by the security module. Hence, the security component depends on the address component.

7. Use Cases

7.1 Transaction

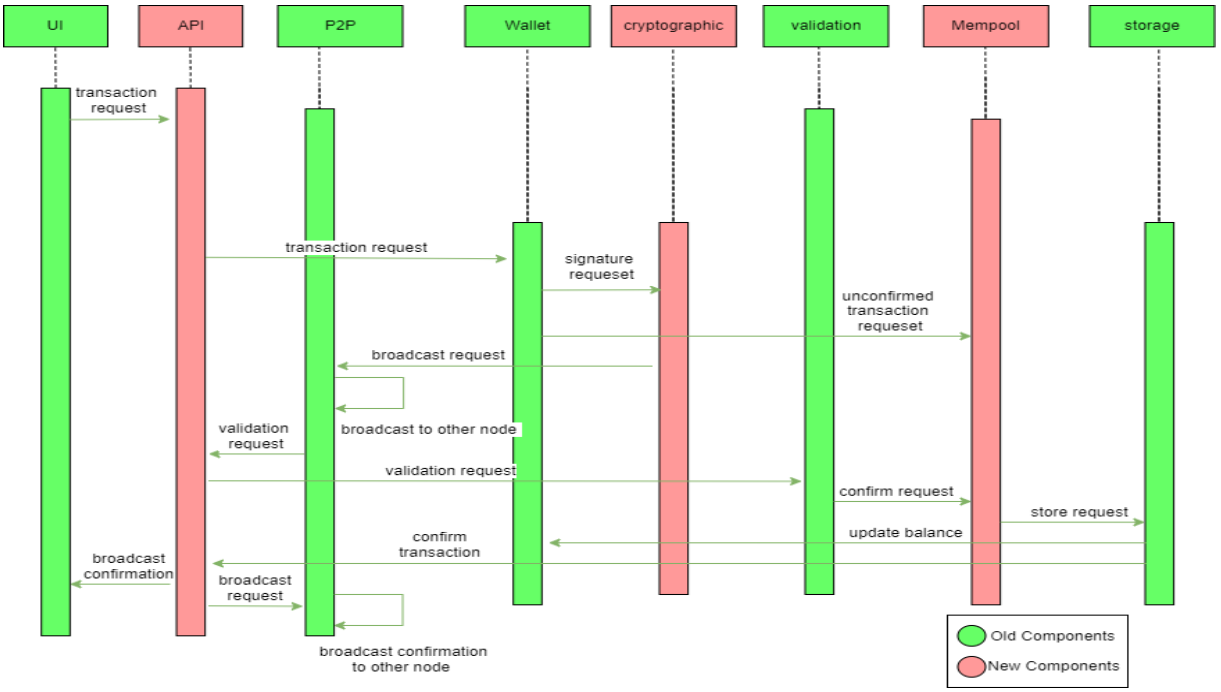

Figure 5. Sequence Diagram of Transaction

The first use case shows a transaction scenario (Figure 5) in which the user first interacts with the API through the UI, i.e., sends a transaction request to the API, which then forwards the request to the wallet component. At this point, the wallet component will do two things: first, it will make a signature request through the cryptographic component, and second, it will send the unverified transaction request to the mempool. After the signature request is done, the cryptographic component will send a broadcast request to the P2P network, where the P2P will broadcast the message to other nodes, and then send a validation request to the API, which will forward the request to the validation component, which will send the confirmation message to the mempool after a successful validation, so that the mempool will update the previous unconfirmed request. and save it to the storage component. After saving, the balance of the wallet will be updated and the completion message will be sent to the API, which will return the broadcast confirmation message to the UI for the user to check, and send the broadcast request to the P2P network to broadcast it to other nodes. The above is the complete flow of this scenario.

7.2 Mining

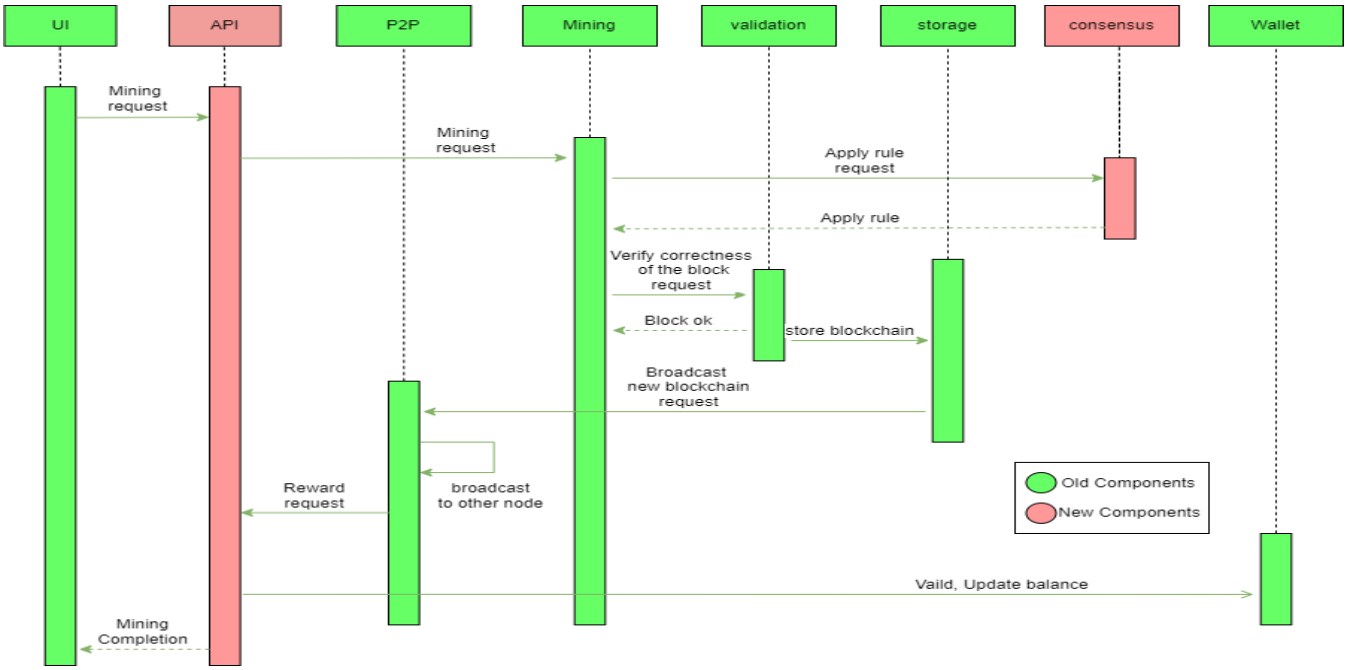

Figure 6. Sequence Diagram of Solo-Mining

The second use case presents the solo-mining scenario (Figure 6). The user can perform the mining operation through the UI. Then the mining request is dispatched to Mining through API. The consensus includes the mining rules and some algorithms inside. When mining is running, Mining will request these regulations from Consensus, and load the terms. After Mining creates a new block, it will request Validation to verify the validity of transactions and blocks. For example, the electronic signature is correct, the transaction is correctly formatted and the block is correctly constructed and linked.

After successful verification, the new blockchain and other data are stored inside Storage and then notified to other users via the P2P network. At the same time, the P2P network will also request a reward from the API because a new block is created, and then the API lets the Wallet update the balance

8. Lessons Learned

After a comprehensive analysis of the Bitcoin architecture, our team gained insight into the design, implementation, and operation of decentralized cryptocurrency systems. We examined the difference between conceptual and concrete architectures to gain insight into the reasons behind Bitcoin's design and complex implementation. Modular design is important in complex systems such as Bitcoin, separating components and functionality, improving ease of system management, maintenance, and component reusability. We also realized the importance of continuous improvement and lessons learned to optimize our analysis methods and reporting structure.

9. Conclusions

In summary, by examining the selected subsystems and comparing them to the conceptual architecture in A1, we found a number of divergences that reveal the intricate nature of implementing a decentralized system like Bitcoin. By using the Understand tool and examining the source code, we gained a deeper understanding of how the Bitcoin system works in the real world, including specific dependencies, data structures, and algorithms used in selected subsystems. Our analysis helps enhance our understanding of blockchain technology and decentralized systems, providing a solid foundation for future research and development in this exciting field.

10. Data Dictionary

Wallet: A digital wallet used to store, send, and receive bitcoins, consists of two components: a public address and a private key.
Signature: A mathematical proof for ownership of a particular Bitcoin address.
Mining: The process of adding new transactions to the Bitcoin blockchain and receiving newly minted bitcoins.
Miner: A participant in the process of adding new transactions to the Bitcoin blockchain.
Consensus: The agreement among all nodes in the network about the state of the blockchain.
Block: A collection of transactions that have been validated by miners and added to the blockchain.
Block Chain: A public ledger that contains a continuously growing list of all transactions that have ever occurred on the Bitcoin network.
Address: A unique identifier that represents a destination for Bitcoin transactions.
TxIn: The input part of a Bitcoin transaction.
TxOut: The output part of a Bitcoin transaction.
Node: A computer or device that is connected to the Bitcoin network and helps to maintain the decentralized blockchain.
Peer: Any other node on the Bitcoin network that a node is connected to.
Public key: Generate Bitcoin addresses, which are used to send and receive Bitcoin transactions.
Private key: Derived from the private key through a mathematical process called Elliptic Curve Cryptography (ECC).

Reference

[1] Bitcoin. (n.d.). *Bitcoin/Bitcoin: Bitcoin Core Integration/Staging tree*. GitHub. Retrieved March 23, 2023, from https://github.com/bitcoin/bitcoin

[2] *P2P network*. Bitcoin. (n.d.). Retrieved March 23, 2023, from https://developer.bitcoin.org/devguide/p2p_network.html