Bitcoin Transaction Creation and Validation Report

Akella Akhila, 230001005
Kommireddy Jayanthi, 230001041
Parimi Sunitha, 230001061

Legacy (P2PKH) Transactions

1. Introduction

In this section, we focus on Legacy (P2PKH) Bitcoin transactions. P2PKH stands for "Pay-to-PubKey-Hash," which is the traditional Bitcoin address format used in many Bitcoin transactions. The goal of this report is to explain the transaction creation and validation process, followed by an in-depth analysis of the script structures involved and the size of a typical P2PKH transaction.

2. Transaction Creation for P2PKH

2.1 Wallet Creation and Address Generation

Connecting to Bitcoin Core: The Python script uses the `bitcoinrpc` library to connect to the Bitcoin Core node.
Example:

```
from bitcoinrpc.authproxy import AuthServiceProxy
rpc_connection =
AuthServiceProxy(f"http://{RPC_USER}:{RPC_PASSWORD}@{RPC_HOST}:{
RPC_PORT}")
```

Address Generation: A Legacy (P2PKH) address is generated using the `getnewaddress` RPC command. This will allow us to create new addresses that follow the P2PKH format.
Example:

```
address_A = rpc_connection.getnewaddress("", "legacy")
address_B = rpc_connection.getnewaddress("", "legacy")
address_C = rpc_connection.getnewaddress("", "legacy")
```

2.2 Transaction Workflow

Funding Address A: Address A is funded with a specific amount of BTC (e.g., 10 BTC) using the `sendtoaddress` RPC command.

Example:
`txid_fund_A = rpc_connection.sendtoaddress(address_A, 10)`

- **TXID for Transaction A to B**: The transaction ID (TXID) of the funding transaction is saved for later reference. This is the starting point for creating subsequent transactions.

Transaction Creation: After the funding transaction is confirmed, a raw transaction is created to send funds from Address A to Address B using the `createrawtransaction` RPC command.
Example:
# Create raw transaction for A → B
raw_tx = rpc_connection.createrawtransaction([{"txid": txid, "vout": vout}], outputs)

Signing,Decoding and Broadcasting: The raw transaction is signed and broadcasted to the network using the `signrawtransactionwithwallet`,`decoderawtransaction` and `sendrawtransaction` commands.
Example:
signed_tx = rpc_connection.signrawtransactionwithwallet(raw_tx)
decoded_raw_tx = rpc_connection.decoderawtransaction(signed_tx['hex'])
broadcast_txid = rpc_connection.sendrawtransaction(signed_tx['hex'])

3. Script Analysis for P2PKH Transactions

3.1 Locking Script (ScriptPubKey)

Structure:
`OP_DUP OP_HASH160 <Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG`

- **OP_DUP**: Duplicates the top item on the stack (used for handling the public key).
- **OP_HASH160**: Hashes the public key using both SHA-256 and RIPEMD-160.
- **OP_EQUALVERIFY**: Verifies that the hash of the public key matches the one provided.
- **OP_CHECKSIG**: Verifies the signature using the public key.

3.2 Unlocking Script (ScriptSig)

Structure:
`<Signature> <Public Key>`

- The unlocking script contains the signature and public key required to unlock the funds for spending.

4. Transaction Size Analysis for P2PKH

- **Input Size**: Approx. 148 bytes (includes ScriptSig and metadata).
- **Output Size**: Approx. 34 bytes (includes ScriptPubKey and amount).
- **Total Transaction Size**: A typical single-input, single-output P2PKH transaction is approximately 226 bytes.

5. Workflow Between Transactions (A → B → C)

To demonstrate the use of the transaction from A to B as an input for the next transaction (B to C):

1. **Transaction A to B**: The first transaction (A → B) was created and broadcasted as shown above, with the `txid` representing the TXID of the completed transaction.

2. **Using Transaction A to B as Input for Transaction B to C**:

   - After Address B receives the BTC, it will be used as an input for the next transaction (B → C).

   - The UTXO from Address B is selected using the `listunspent` command.

Example:
```
utxos_B = rpc_connection.listunspent(1, 9999999, [address_B])
```

3. **Transaction Creation from B to C**: A new raw transaction is created to send funds from Address B to Address C.
   Example:
   ```
   raw_tx_B_to_C = rpc_connection.createrawtransaction([{"txid": utxo_B['txid'], "vout": utxo_B['vout']}], outputs_B_to_C)
   ```

6. Decoded Scripts for Both Transactions

Here are the decoded scripts for the two transactions:

Transaction A → B:

Locking Script (ScriptPubKey):

`OP_DUP OP_HASH160 <Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG`

Unlocking Script (ScriptSig):

`<Signature> <Public Key>`

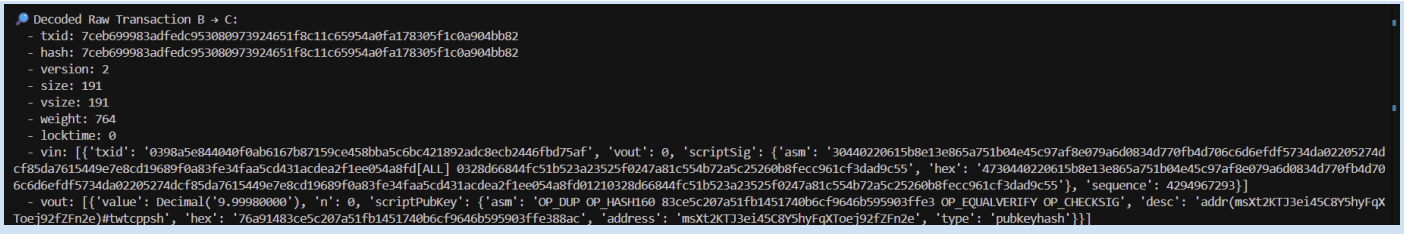
Transaction B → C:

Locking Script (ScriptPubKey):

`OP_DUP OP_HASH160 <Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG`

Unlocking Script (ScriptSig):

`<Signature> <Public Key>`



7. Script Structure and Challenge-Response Validation

7.1 Challenge Script (Locking Script)

The locking script (ScriptPubKey) is the "challenge" in the Bitcoin script execution. For both P2PKH transactions (A → B and B → C), the challenge is to verify that the public key hash in the script matches the public key provided by the unlocking script (ScriptSig) and that the signature is valid.

7.2 Response Script (Unlocking Script)

The unlocking script (ScriptSig) provides the response to the challenge. It contains the public key and signature that are used to verify the ownership of the funds.

- **Challenge**: The locking script requires that the public key hash matches the provided public key.
- **Response**: The unlocking script provides the signature and public key to prove ownership of the funds.

8. Bitcoin Debugger Execution and Screenshots

The Bitcoin Debugger can be used to analyze the scripts and validate the challenge-response mechanism.

- **Steps in Bitcoin Debugger**:
    1. **Input the transaction**: Use the `decoderawtransaction` command to decode the raw transaction.
    2. **Validate Locking and Unlocking Scripts**: Use the `decodescript` command to decode and validate the locking and unlocking scripts.

- **Screenshots**:

  Transaction from A to B

Transaction from B to C

Segwit (P2SH-P2WPKH) Transactions

1. Introduction

Segregated Witness (SegWit) is an upgrade to the Bitcoin protocol that improves scalability and transaction efficiency. By separating the signature data (referred to as the "witness") from the transaction data, SegWit transactions allow more data to fit into a single block, resulting in smaller transaction sizes and reduced fees. SegWit transactions, specifically **P2SH-P2WPKH**, also offer improved security and flexibility compared to the traditional **P2PKH** format.

This section will walk through the transaction creation process, script analysis, and explain how SegWit (P2SH-P2WPKH) transactions work, including the transaction workflow, challenge-response script structure, and the validation process.

2. Transaction Creation for SegWit (P2SH-P2WPKH)

2.1 Wallet Creation and Address Generation

Connecting to Bitcoin Core: The Python script establishes a connection to the Bitcoin Core node via the `bitcoinrpc` library. This connection allows us to interact with the Bitcoin network to generate SegWit addresses and create transactions.

Example code to connect:

```
from bitcoinrpc.authproxy import AuthServiceProxy
rpc_connection =
AuthServiceProxy(f"http://{RPC_USER}:{RPC_PASSWORD}@{RPC_HOST}:{RPC_PORT}")
```

Address Generation: A SegWit (P2SH-P2WPKH) address is generated using the `getnewaddress` RPC command. This address is specifically designed to be compatible with the SegWit structure.

Example of generating a SegWit address:

```
address_A = rpc_connection.getnewaddress("", "p2sh-segwit")
address_B = rpc_connection.getnewaddress("", "p2sh-segwit")
address_C = rpc_connection.getnewaddress("", "p2sh-segwit")
```

2.2 Transaction Workflow

Funding Address A: Just as with traditional Bitcoin transactions, SegWit transactions require funding. Address A is funded with a specific amount of BTC (e.g., 10 BTC) via the `sendtoaddress` RPC command.

Example code for funding:

```
txid_fund_A = rpc_connection.sendtoaddress(address_A, 10)
```

Transaction Creation (A to B): Once the funding transaction is confirmed, a raw transaction is created to send funds from Address A to Address B. The SegWit address is used for the output.

Example code for creating a raw transaction:

```
raw_tx = rpc_connection.createrawtransaction([{"txid": txid, "vout":
vout}], outputs)
```

Signing,Decoding and Broadcasting: The raw transaction is signed with the private key of Address A and broadcast to the Bitcoin network.

Example code to sign and send:

```
signed_tx = rpc_connection.signrawtransactionwithwallet(raw_tx)
decoded_raw_tx = rpc_connection.decoderawtransaction(signed_tx['hex'])
broadcast_txid = rpc_connection.sendrawtransaction(signed_tx['hex'])
```

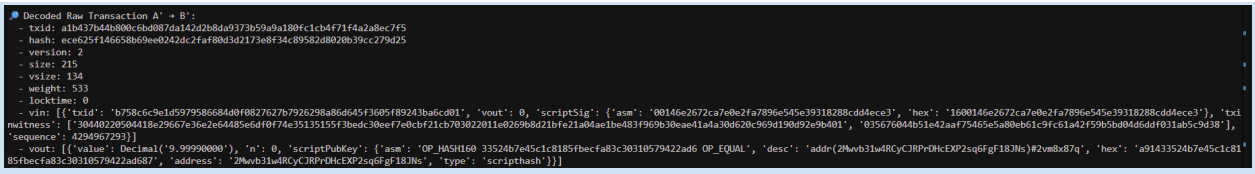
Transaction Creation (B to C): Once the transaction from A to B is confirmed, Address B can use this transaction as input for a new transaction (from B to C).

Example code for creating a transaction from B to C:

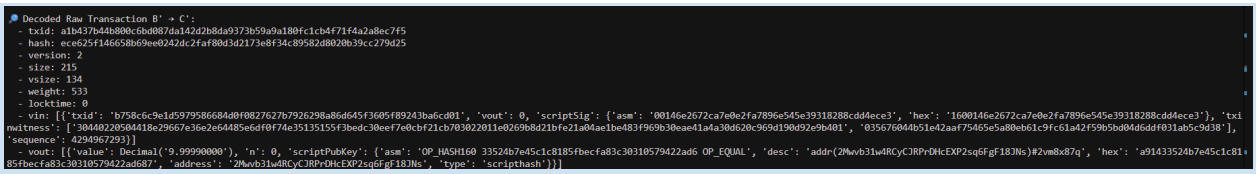
```
txid_B = utxo_B['txid']
raw_tx_B_to_C = rpc_connection.createrawtransaction([{"txid": txid_B,
"vout": vout_B}], outputs_B_to_C)
signed_tx_B_to_C =
rpc_connection.signrawtransactionwithwallet(raw_tx_B_to_C)
broadcast_txid_B_to_C =
rpc_connection.sendrawtransaction(signed_tx_B_to_C['hex'])
```

Decoded Scripts for Both Transactions

`Transaction from A to B`



`Transaction from B to C`



3. Script Analysis for P2SH-P2WPKH Transactions

3.1 Decoded Scripts for Both Transactions

Let's break down the scripts for both transactions:

1. **Transaction from A to B** (P2SH-P2WPKH):
    - **Locking Script (ScriptPubKey)**:
      `OP_HASH160 <Public Key Hash> OP_EQUAL`
2. This script locks the output by requiring the public key hash to match the specified value.
    - **Unlocking Script (ScriptSig)**:
      `0 <Witness Script>`
3. The unlocking script for SegWit transactions contains a witness that includes the signature and public key. The '0' denotes the version number of the SegWit transaction.

4. **Transaction from B to C** (P2SH-P2WPKH):
    - **Locking Script (ScriptPubKey)**:
      `OP_HASH160 <Public Key Hash> OP_EQUAL`
5. The locking script for this transaction is identical to the first one, requiring the public key hash to match the specified value.
    - **Unlocking Script (ScriptSig)**:
      `0 <Witness Script>`
6. Similarly, the unlocking script for this transaction also contains the witness script, which validates the signature and public key.

3.2 Challenge-Response Script Structure

In Bitcoin transactions, the **challenge** is typically the condition that must be met for the transaction to be valid (e.g., the public key hash and the signature). The **response** is the data provided by the sender to prove they can fulfill the challenge.

- **Locking Script** (Challenge): The locking script is the challenge part of the transaction. In P2SH-P2WPKH, the challenge is to provide the public key hash that matches the one stored in the script. The operation `OP_EQUAL` validates whether the provided public key hash matches the stored value.
- **Unlocking Script** (Response): The unlocking script is the response that provides the necessary data to fulfill the challenge. For SegWit transactions, the response includes a witness script with the signature and public key. The signature is validated using the public key to ensure that the transaction is authorized by the rightful owner.

The **challenge-response** structure ensures that only the rightful owner, who has access to the correct signature and public key, can spend the funds locked in the transaction output.

3.3 Validation of the Transaction

The transaction is validated through the execution of the challenge-response scripts:

1. The Bitcoin network verifies the locking script (challenge) to ensure the public key hash matches the expected value.
2. The unlocking script (response) is then checked by validating the signature against the public key. If the signature matches, the funds are unlocked and the transaction is considered valid.

4. Screenshots of Bitcoin Debugger Execution

To further clarify the process, we can provide **screenshots of steps of the Bitcoin debugger** executing the challenge-response validation process. These steps typically involve:

1. Using a tool like **Bitcoin Core's Debugger** to inspect the raw transaction and view the decoded scripts.

Transaction from A to B

Transaction from B to C

5. Conclusion

SegWit (P2SH-P2WPKH) transactions represent a more efficient and scalable method for conducting Bitcoin transactions. By separating the witness data (signature and public key) from the main transaction body, SegWit reduces transaction sizes and improves overall performance on the Bitcoin network.

Key benefits of SegWit transactions:

- **Reduced Transaction Size**: SegWit transactions store the signature and public key in the witness data structure, reducing the overall size of the transaction.

- **Lower Fees**: Smaller transactions lead to lower fees due to reduced space requirements in each block.

- **Increased Scalability**: SegWit transactions improve the scalability of the Bitcoin network by allowing more transactions to fit into each block.

In the next section, we will compare **SegWit (P2SH-P2WPKH)** transactions with **Legacy (P2PKH)** transactions, focusing on transaction sizes, script structures, and overall efficiency.

Comparison of P2PKH and P2SH-P2WPKH Transactions

1. Transaction Size Comparison

The size of a transaction is critical to understanding its impact on network efficiency, transaction fees, and scalability. Let's compare the sizes of P2PKH (legacy) and P2SH-P2WPKH (SegWit) transactions.

P2PKH Transaction Size (Legacy)

- **Input Size**: A P2PKH input typically occupies around **148 bytes**. This includes the **ScriptSig**, which contains the signature and public key, as well as other metadata associated with the transaction.
- **Output Size**: The output size is approximately **34 bytes** for a single output, which includes the **ScriptPubKey** and the amount.
- **Total Transaction Size**: A typical P2PKH transaction, with one input and one output, is around **226 bytes**.

P2SH-P2WPKH Transaction Size (SegWit)

- **Input Size**: A SegWit input (P2SH-P2WPKH) is smaller, with an average size of around **71 bytes**. This is because SegWit transactions separate the witness data (signature and public key) from the main transaction body, storing it in a separate structure.
- **Output Size**: The output size is similar to P2PKH, approximately **34 bytes**.
- **Total Transaction Size**: A typical SegWit (P2SH-P2WPKH) transaction, with one input and one output, is around **160 bytes**, which is significantly smaller than the equivalent P2PKH transaction.

Size Comparison Summary

Type of Transaction	Input Size (bytes)	Output Size (bytes)	Total Size (bytes)
P2PKH (Legacy)	148	34	~226
P2SH-P2WPKH (SegWit)	71	34	~160

- **P2PKH** transactions are **larger** due to the inclusion of both the public key and signature in the unlocking script (ScriptSig), alongside other metadata.
- **P2SH-P2WPKH** transactions are **smaller** because they store the signature and public key separately in the witness data structure, reducing the main transaction size.

2. Script Structure Comparison

The script structure of Bitcoin transactions determines how they validate inputs and outputs. In this section, we'll compare the **locking** and **unlocking** scripts of P2PKH (legacy) and P2SH-P2WPKH (SegWit) transactions.

P2PKH (Legacy) Script Structure

1. **Locking Script (ScriptPubKey)**:
    - Structure:
      `OP_DUP OP_HASH160 <Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG`
    - **Explanation**:
        - `OP_DUP`: Duplicates the top item on the stack (the public key).
        - `OP_HASH160`: Applies both SHA-256 and RIPEMD-160 hash functions to the public key.
        - `<Public Key Hash>`: The hash of the public key that will be compared to the public key hash provided in the transaction.
        - `OP_EQUALVERIFY`: Ensures that the hash of the public key matches the expected value.
        - `OP_CHECKSIG`: Verifies the signature against the public key.

2. **Unlocking Script (ScriptSig)**:
    - Structure:
      `<Signature> <Public Key>`
    - **Explanation**:
        - The unlocking script contains the **signature** and **public key** required to unlock the transaction and spend the funds.

P2SH-P2WPKH (SegWit) Script Structure

1. **Locking Script (ScriptPubKey)**:
    - Structure:
      `OP_HASH160 <Public Key Hash> OP_EQUAL`
    - **Explanation**:
        - `OP_HASH160`: Hashes the public key.
        - `<Public Key Hash>`: The public key hash is compared against the value in the locking script.
        - `OP_EQUAL`: Verifies that the provided public key hash matches the one stored in the script.

2. **Unlocking Script (ScriptSig)**:
    - Structure:
      `0 <Witness Script>`
    - **Explanation**:
        - `0`: Denotes the version number of the SegWit transaction.
        - `<Witness Script>`: Contains the **signature** and **public key** in the witness data structure.

Challenge-Response Script

- **P2PKH (Legacy)**:
    - **Challenge (Locking Script)**: The challenge is to match the provided public key hash with the one stored in the locking script, and the signature must validate against the public key.
    - **Response (Unlocking Script)**: The response includes the signature and public key that are verified by the Bitcoin network to prove ownership of the funds.

- **P2SH-P2WPKH (SegWit)**:
    - **Challenge (Locking Script)**: The challenge is simpler and consists of verifying that the public key hash matches the one stored in the script using the `OP_EQUAL` operator.
    - **Response (Unlocking Script)**: The response contains the **witness script**, which includes the signature and public key, stored separately from the main transaction body.

3. Why Are SegWit Transactions Smaller?

SegWit transactions are smaller due to their design, which separates the **witness data** (the signature and public key) from the main transaction body. This results in a smaller size for each input, as the witness data is stored separately and not included in the main body of the transaction. The key reasons why SegWit transactions are smaller are as follows:

1. **Witness Data**: In SegWit transactions, the signature and public key are moved to the **witness** section, which is separate from the regular transaction body. This allows the rest of the transaction to be more compact, reducing the overall size.
2. **Separation of Witness**: By placing the signature and public key outside the main transaction body, SegWit reduces the amount of data included in the **ScriptSig** and **ScriptPubKey**, resulting in smaller transaction sizes.
3. **Improved Efficiency**: The separation of the witness data allows for **more efficient block space utilization**, meaning more transactions can fit into a single block. This improves the scalability of the Bitcoin network, as blocks can accommodate more transactions without exceeding the block size limit.

4. Benefits of Smaller SegWit Transactions

Smaller SegWit transactions offer several key benefits:

1. **Lower Transaction Fees**: Since SegWit transactions are smaller, they require less space in a block, which results in lower transaction fees. Bitcoin transaction fees are typically based on the size of the transaction in bytes, so smaller transactions are less expensive to process.
2. **Increased Scalability**: SegWit allows more transactions to fit into each block, improving the overall scalability of the Bitcoin network. This helps alleviate congestion and improves the user experience by reducing transaction delays.
3. **Faster Processing**: SegWit transactions are processed more efficiently due to their smaller size. This leads to faster transaction verification and confirmation times.
4. **Enhanced Security**: SegWit also enhances security by fixing certain vulnerabilities, such as transaction malleability. This ensures that transactions cannot be altered once they are broadcast to the network, increasing the reliability of the system.

5. Conclusion

The comparison between **P2PKH** (Legacy) and **P2SH-P2WPKH** (SegWit) transactions highlights the improvements brought by the SegWit protocol. While both transaction types achieve the same goal—moving Bitcoin from one address to another—the key differences lie in their structure, size, and efficiency:

- **P2PKH** transactions are **larger** because they include both the signature and public key in the unlocking script, leading to higher transaction sizes and fees.
- **P2SH-P2WPKH** transactions, as part of the **SegWit** upgrade, are **smaller** due to the separation of witness data (signature and public key) from the main transaction body. This results in lower transaction sizes, reduced fees, and improved scalability.

SegWit provides significant benefits in terms of **transaction efficiency**, **lower fees**, and **network scalability**, making it the preferred choice for modern Bitcoin transactions.