Bitcoin Scripting Report

A Comparative Analysis of
Legacy and SegWit Transactions

This report presents an in-depth analysis of Bitcoin transaction scripts, comparing the traditional Legacy (P2PKH) format with the newer Segregated Witness (P2SH-P2WPKH) format. Bitcoin's foundation relies on its scripting system—a stack-based language that determines transaction validity through cryptographic challenges and responses.

Key Scripting Components

The scripting system consists of two primary components:

- **ScriptPubKey (Locking Script)**: Placed on outputs, defining conditions required to spend bitcoins
- **ScriptSig (Unlocking Script)**: Provided by the spender to satisfy the conditions in the ScriptPubKey

The objectives of this assignment were to:

- Create and analyze Legacy (P2PKH) transactions in a controlled regtest environment
- Create and analyze SegWit (P2SH-P2WPKH) transactions in the same environment
- Compare transaction structures, sizes, and scripts
- Understand the benefits and implications of the SegWit upgrade

All transactions were created in Bitcoin Core's regtest mode, which provides a controlled environment for testing without requiring real bitcoins. The following configuration was used:

Bitcoin Core Configuration (bitcoin.conf)

```
# Network settings
regtest=1

# RPC settings
server=1
rpcuser=[username]
rpcpassword=[password]
rpcallowip=127.0.0.1
rpcport=18443

# Fee settings
paytxfee=0.0001
fallbackfee=0.0002
mintxfee=0.00001
txconfirmtarget=1
```

We implemented the assignment using Node.js with the following components:

- **helpers.js**: Core utilities for interacting with Bitcoin Core through RPC
- **legacyTransactions.js**: Implementation of Legacy transaction flow
- **segwitTransactions.js**: Implementation of SegWit transaction flow

For each transaction type, we followed these steps:

1. Create three addresses (A, B, C for Legacy; A', B', C' for SegWit)
2. Fund the first address (A or A')
3. Create a transaction from first to second address (A→B or A'→B')
4. Create a transaction from second to third address (B→C or B'→C')
5. Analyze the transaction scripts and structures

1 Legacy (P2PKH) Transactions

1.1 Transaction Flow Overview

Legacy transactions use the Pay-to-Public-Key-Hash (P2PKH) format, which is the traditional Bitcoin address format. Our transaction flow involved:

1.2 Transaction Details

Transaction IDs

```
Funding TX:
    e186bba556710ba24210a770757ac8fb04effa09d7f862c9396c601d07f5f9fd
A to B TX:
    d2153ddb01557451182abef28049c83905f832bb781da6767552a9f8aebe440d
B to C TX:    7
    da38581369c7445becd0622b98f6ff6dec1ba84efdc29c3412b731401807734
```

1.3 Script Analysis

1.3.1 Locking Script (ScriptPubKey) for Address B

P2PKH Locking Script

```
OP_DUP OP_HASH160 7ddc450c8dd929b04c3aed59822d6ddca4e84a80
    OP_EQUALVERIFY OP_CHECKSIG
```

This script implements the following logic:

1. **OP_DUP**: Duplicates the provided public key on the stack
2. **OP_HASH160**: Applies SHA-256 followed by RIPEMD-160 to the public key
3. **7ddc450c...**: Pushes the expected public key hash (the "address" without prefix/checksum)
4. **OP_EQUALVERIFY**: Verifies the hash equality and terminates if false
5. **OP_CHECKSIG**: Verifies the signature against the public key

1.3.2 Unlocking Script (ScriptSig) in B to C Transaction

P2PKH Unlocking Script

```
<Signature> <PublicKey>
```

From our transaction:

```
3044022065565eb1aaab2d5e434ce9c03c9694ac06264b7d92451838909d7454a38b405d
0220082f5abdbad3ecac3ad8552524f5ed0dda03956d7a94d868d20f047e562eabbe[ALL]
0347132e9f4e9b9ad8665e14d856e76373e0dfb60fe946b1c0048f9fc8d9a807a6
```

This unlocking script provides two critical pieces of data:

- The digital signature (3044022065565e...), which proves ownership of the private key
- The public key (0347132e9f4e...), which when hashed should match the hash in the locking script

1.4 Script Execution Process

When a Bitcoin node validates the transaction, it executes the combined scripts step-by-step:

ScriptSig: <Signature> <PublicKey>

↓

ScriptPubKey: OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

↓

Combined: <Signature> <PublicKey> OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

The execution steps and stack state:

P2PKH Script Execution Steps

Operation	Stack (after operation)
Initial	[]
Push ⟨Signature⟩	[⟨Signature⟩]
Push ⟨PublicKey⟩	[⟨PublicKey⟩, ⟨Signature⟩]
OP_DUP	[⟨PublicKey⟩, ⟨PublicKey⟩, ⟨Signature⟩]
OP_HASH160	[⟨PubKeyHash'⟩, ⟨PublicKey⟩, ⟨Signature⟩]
Push ⟨PubKeyHash⟩	[⟨PubKeyHash⟩, ⟨PubKeyHash'⟩, ⟨PublicKey⟩, ⟨Signature⟩]
OP_EQUALVERIFY	[⟨PublicKey⟩, ⟨Signature⟩] (if hashes match)
OP_CHECKSIG	[TRUE/FALSE]

2 SegWit (P2SH-P2WPKH) Transactions

2.1 Transaction Flow Overview

SegWit transactions use a nested Pay-to-Script-Hash wrapping a Witness program. Our transaction flow involved:

2.2 Transaction Details

Transaction IDs

```
Funding TX:
    e2ec9d440eff5b81d0b9fec11546f05858416b442e9eb83d77085548d85261b3
A' to B' TX:
    a4a164ebdbd2772d098d9ae86afc3976778f15b5cc92a5ae9736d8e354d273eb
B' to C' TX:
    a25e8dd4bd3973e059be21b36c628cd35dc4226ab02156646ba7719270350a09
```

2.3 Script Analysis

2.3.1 Locking Script (ScriptPubKey) for Address B'

P2SH-P2WPKH Locking Script

```
OP_HASH160 83995dc684b6bb992ebd4d974c4cce460ef568e0 OP_EQUAL
```

This is a standard P2SH script that:

1. **OP_HASH160**: Applies SHA-256 followed by RIPEMD-160 to the redeem script
2. **83995dc6...**: Pushes the expected script hash
3. **OP_EQUAL**: Verifies the hash equality

2.3.2 Unlocking Script (ScriptSig) in B' to C' Transaction

P2SH-P2WPKH Unlocking Script

```
0014ffb3bc98fa9116330fc6a3616604a2a3e3a6db45
```

This is the redeem script containing the witness program. Breaking it down:

- **00**: Version 0 witness program

- **14**: Push the next 20 bytes onto the stack
- **ffb3bc98...**: The public key hash (20 bytes)

2.3.3 Witness Data

SegWit Witness Data

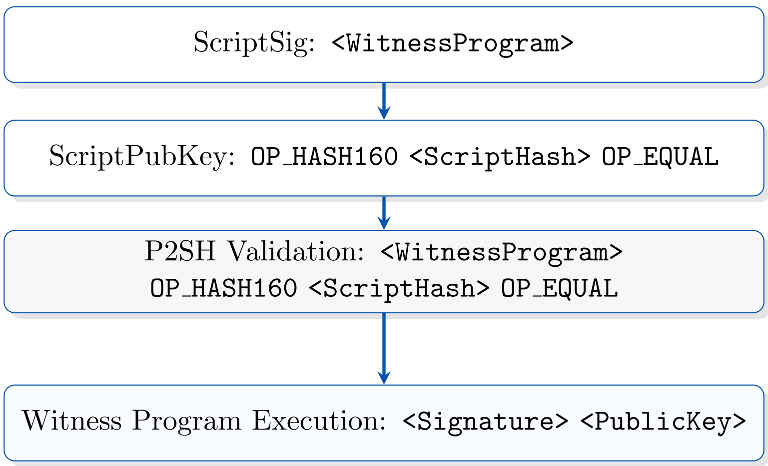
```
[
  "3044022061635528
   fd38990b16b91a6bb3ebe121d49c1b37523d368eca35803b5be531e9
  022018
   f75609b0852660197cc586ab011e3115fad5a71577b7c93e2af1d21b2b795c01
   ",
  "03
   bb3a761eb5803a9e175dbf0da9c5f86497be72a08f5800d50119da59980c0f96"
]
```

The witness data contains:

- The signature (304402206163...)
- The public key (03bb3a761eb5...)

2.4 Script Execution Process

The P2SH-P2WPKH validation occurs in multiple phases:



P2SH-P2WPKH Script Execution Steps

Phase 1: P2SH Validation	
Initial	[]
Push ⟨WitnessProgram⟩	[⟨WitnessProgram⟩]
OP_HASH160	[⟨WitnessProgram_hash⟩]
Push ⟨ScriptHash⟩	[⟨ScriptHash⟩, ⟨WitnessProgram_hash⟩]
OP_EQUAL	[TRUE] (if hashes match)
Phase 2: Witness Program Execution	
Initial	[]
Push ⟨Signature⟩ (from witness)	[⟨Signature⟩]
Push ⟨PublicKey⟩ (from witness)	[⟨PublicKey⟩, ⟨Signature⟩]
Execute P2WPKH logic	[TRUE/FALSE]

3 Comparative Analysis

3.1 Transaction Size and Weight

Transaction Type	Raw Size	vSize	Weight	Fee Benefit
Legacy (P2PKH) B to C	225 bytes	225 bytes	900	-
SegWit (P2SH-P2WPKH) B' to C'	247 bytes	166 bytes	661	26.22%

Table 1: Transaction Size and Weight Comparison

Size Observations

Note that:

- The raw size of the SegWit transaction is actually larger (247 vs 225 bytes)
- However, the virtual size (vSize) is significantly smaller (166 vs 225 bytes)
- This results in a 26.22% reduction in fee-calculating size

The vSize is calculated as (weight + 3) / 4, reflecting how witness data is discounted for fee calculations.

3.2 Script Structure Comparison

3.3 Key Differences

1. **Script Location**: In Legacy transactions, both signature and public key are in the scriptSig. In SegWit, the signature and public key are moved to the separate witness data.

Component	Legacy (P2PKH)	SegWit (P2SH-P2WPKH)
Address Format	"m" or "n" prefix (testnet)	"2" prefix (testnet P2SH)
Locking Script	OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	OP_HASH160 <ScriptHash> OP_EQUAL
Unlocking Script	<Signature> <PublicKey>	<WitnessProgram>
Witness Data	N/A	<Signature> <PublicKey>

Table 2: Script Structure Comparison

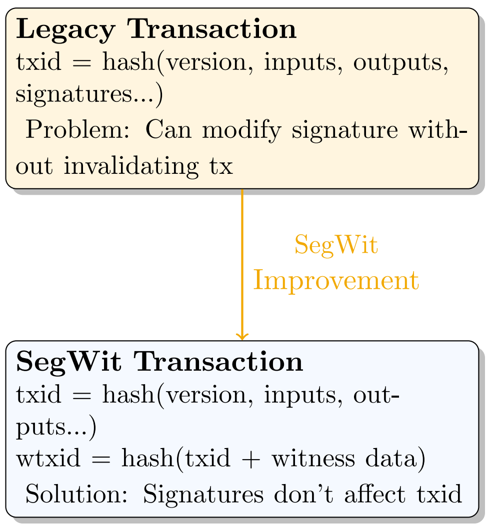
2. **ScriptSig Complexity**: The Legacy scriptSig contains both the signature and public key (typically 106+ bytes). The SegWit scriptSig contains only the witness program (23 bytes), significantly reducing the scriptSig size.
3. **Validation Process**: Legacy scripts execute in a single phase. SegWit scripts execute in two phases: P2SH validation followed by witness program execution.

4 Benefits of SegWit

4.1 Transaction Malleability Resolution

Transaction Malleability

Transaction malleability was a significant issue in Bitcoin prior to SegWit. It occurs when a third party can modify a transaction's signature (without invalidating it) to change the transaction ID (txid) while the transaction is unconfirmed.



Our transactions demonstrate this:

- For Legacy transactions, the scriptSig containing signatures is directly hashed to create the txid

- For SegWit transactions, the witness data (containing signatures) is not included in the txid calculation

4.2 Size Efficiency

SegWit introduces a concept called "weight units" to discount witness data:

- Non-witness data counts as 4 weight units per byte
- Witness data counts as 1 weight unit per byte
- Virtual size = (weight / 4) bytes, which is used for fee calculation

Fee Savings Analysis

In our transactions:

- The Legacy transaction uses 225 bytes with a weight of 900
- The SegWit transaction uses 247 total bytes but only 166 virtual bytes
- This results in approximately 26% lower fees for the same transaction

4.3 Upgrade Path for Bitcoin

SegWit introduced the concept of script versioning through the witness program, enabling:

- Future script upgrades without requiring hard forks
- Backward compatibility with older nodes
- A foundation for advanced features like Taproot

5 Bitcoin Script Debugger Analysis

5.1 Legacy Script Validation

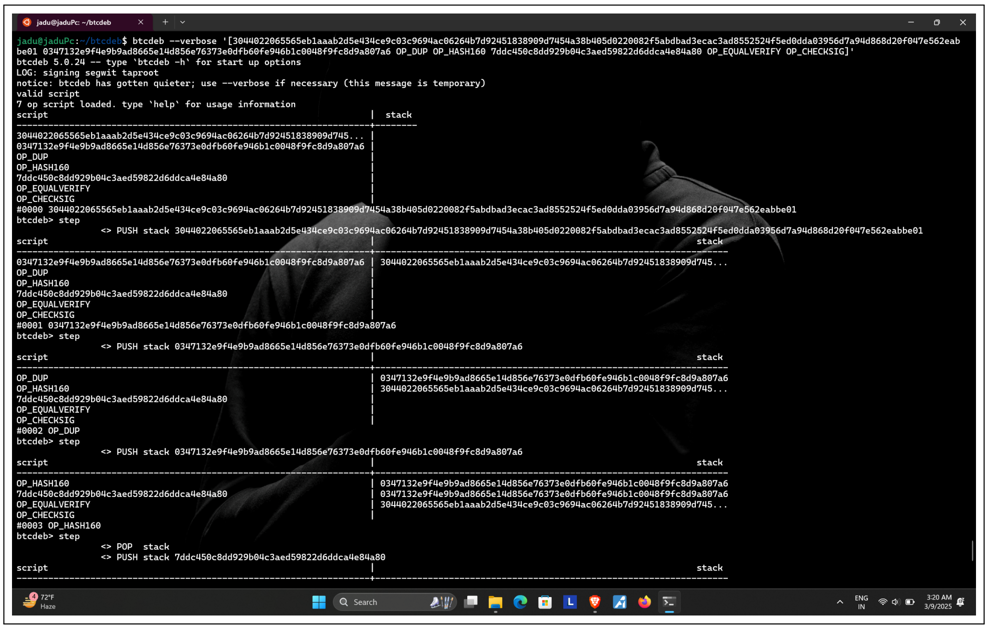


Figure 1: Bitcoin debugger showing Legacy script execution

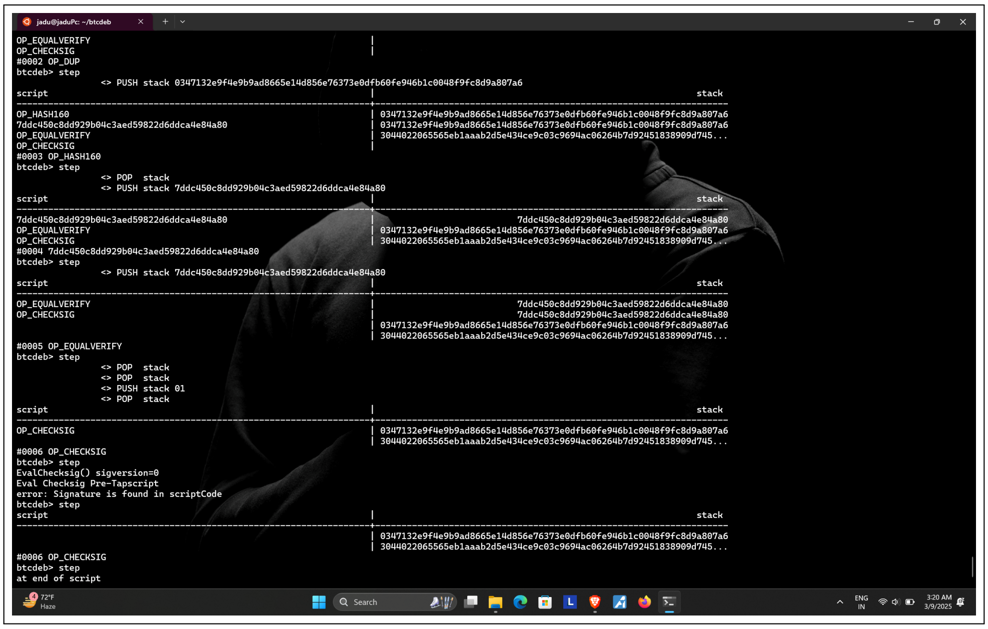


Figure 2: Bitcoin debugger showing Legacy script execution

The debugger shows the execution of the P2PKH script with the following steps:

1. The signature and public key are pushed onto the stack
2. OP_DUP duplicates the public key
3. OP_HASH160 hashes the duplicated public key
4. The hash is compared with the expected PubKeyHash
5. OP_CHECKSIG verifies the signature against the public key

5.2 SegWit Script Validation

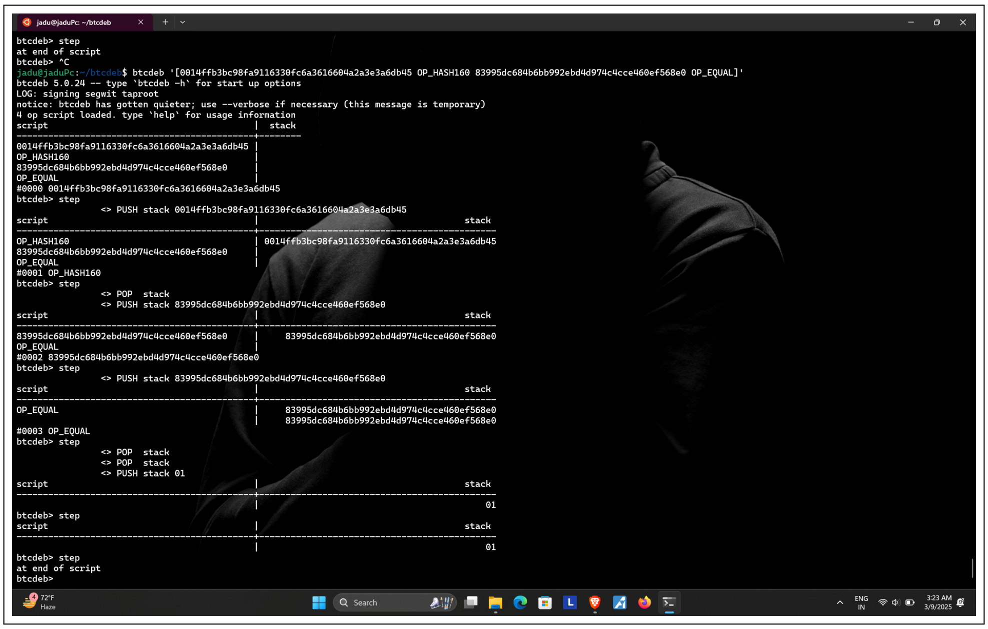


Figure 3: Bitcoin Debugger showing SegWit script execution

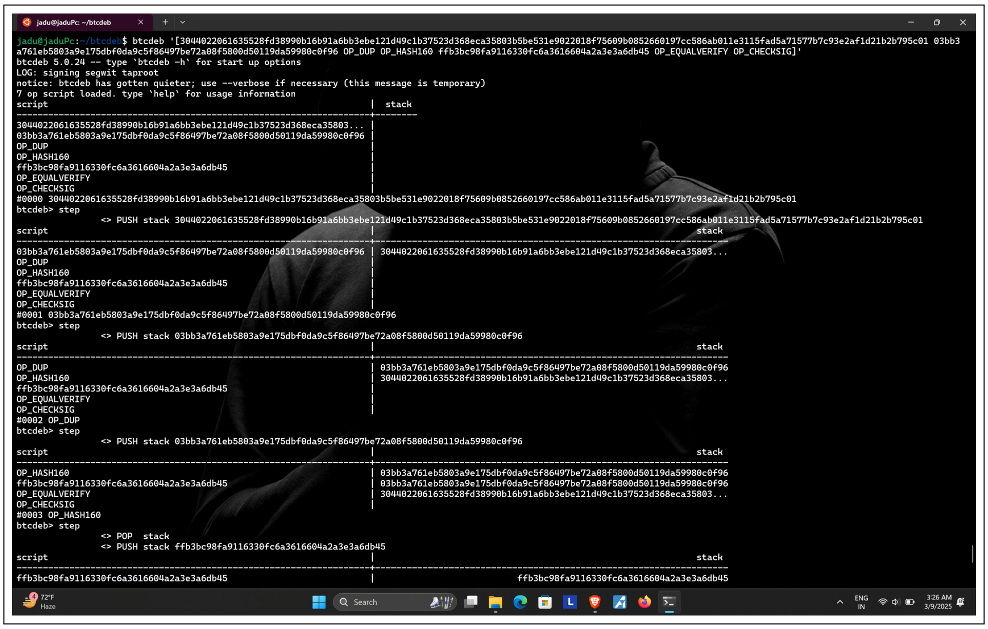


Figure 4: Bitcoin Debugger showing SegWit script execution

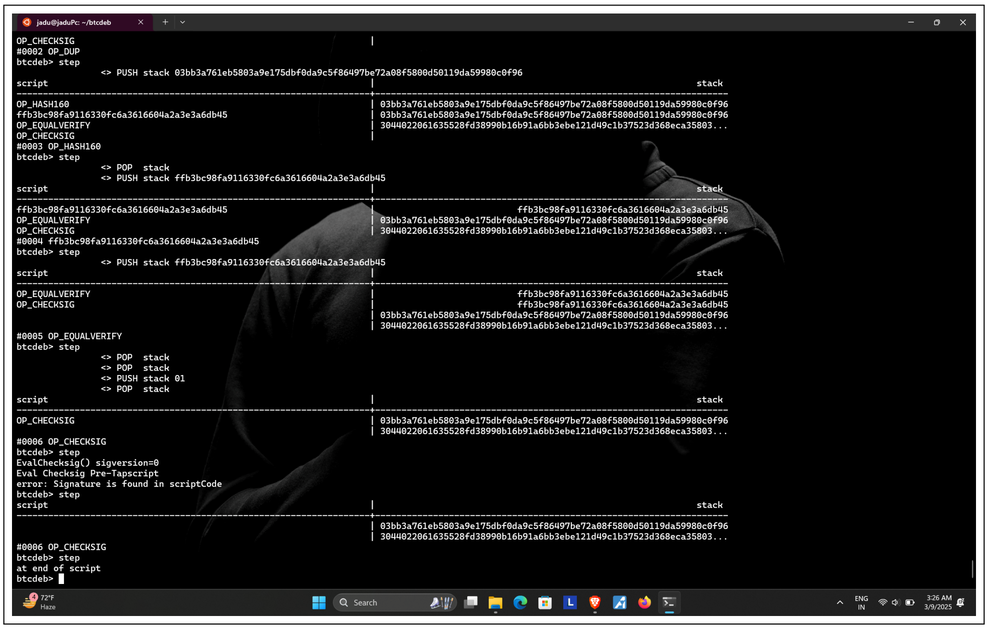


Figure 5: Bitcoin Debugger showing SegWit script execution

The SegWit script execution shows:

1. P2SH validation phase: The witness program is pushed and its hash is compared with the expected script hash

2. Witness program execution: The signature and public key from the witness data are used to validate the transaction