Yenpoint Inc.

BSV Blockchain Library

Python SDK

User Test Report Summary

August 20th, 2024

Prepared for:
BSV Association

Prepared by:
Yenpoint Inc. Ken Sato

About Yenpoint Inc.

Yenpoint Inc. is a pioneering blockchain technology company based in Sendai, Japan, established in August 2020. Under the visionary leadership of CEO Ken Sato, Yenpoint is committed to developing and delivering cutting-edge services centered around blockchain technology.

Our core business spans a wide range of innovations, including the "Yenpoint Button" micropayment solution, the development of NFT marketplaces, next-generation stablecoin infrastructure, SPV wallet development, and our groundbreaking "p2p token verification" technology, which is currently patent-pending.

At Yenpoint, we believe that blockchain technology should be accessible to everyone worldwide. This belief drives our focus on enhancing the scalability and usability of blockchain solutions. Technologically, we leverage advanced tools such as support for the most scalable blockchain, Bitcoin SV and NFT technology utilizing 1 sat Ordinals. Additionally, we actively collaborate with universities on research into Teranode subtrees and conduct user testing on blockchain infrastructure technologies and standard libraries like SPV, all with the goal of refining and implementing these technologies in practical applications.

On the educational front, Yenpoint is dedicated to promoting blockchain knowledge by organizing lectures and seminars, including those at leading science and engineering universities across Japan. Through our joint research initiatives, we aim to disseminate cutting-edge knowledge and technology in the blockchain field.

Our mission, "We improve the quality of the internet by valuing information through blockchain," reflects our commitment to enhancing the value of information on the internet and building a new economic system powered by blockchain technology. We are exploring the vast potential of blockchain across diverse sectors, including gaming, marketplaces, and education, contributing to the growth of the digital economy.

Stay updated with our latest news and developments by following us on LinkedIn. You can also explore our public repositories on GitHub, engage with us directly through our Contact page, or email us at info@yenpoint.jp.

Yenpoint Inc.
1-4-9 enspace Kokubuncho Aoba-ku
Sendai, Miyagi, Japan 980-0803
https://yenpoint.jp/
info@yenpoint.jp

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All work related to this project, carried out by Yenpoint Inc., was completed in full alignment with the statement of work and the mutually agreed-upon project plan.

The assessments of user tests are typically time-constrained and may depend on information provided by the client, its affiliates, or partners. Therefore, the findings detailed in this report should not be regarded as a complete list of all errors, bugs, security vulnerabilities, flaws, or defects within the target system or codebase.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Ken Sato, Project Manager
ken@yenpoint.jp

The following engineering director was associated with this project:

Yosuke Sato, Engineering Director

The following engineer was associated with this project:

Phil Markarian, Engineer

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
July 22-26, 2024	Comparing the *py-sdk* with bsvlib
July 29-August 2, 2024	Testing basic functions and new functions
August 5-9, 2024	Implementing the SDK, Comparing to *Ts-sdk*
August 12-16 2024	Reviewing examples and documentations
August 20, 2024	Delivery of summary report

Executive Summary

Bitcoin Satoshi Vision (BSV) adheres to the original vision of Satoshi Nakamoto, with a focus on aggressively scaling the blockchain for mass adoption, advancing technologies like complex smart contracts, and maintaining extremely low transaction fees.

Historically, the development infrastructure for BSV was initially inherited from the limited, small-block BTC Core development. Following the birth of BSV, many companies within its ecosystem voluntarily developed their own libraries. While these efforts were significant, there was a pressing need for a unified and consistent development environment to support the new generation of scalable blockchain applications such as SPV.

The BSV Association is addressing this need by developing groundbreaking next-generation Bitcoin wallet SDK libraries in Go, TypeScript, and Python. This initiative is part of a larger effort to drive scalable blockchain technology to the global population.

Yenpoint Inc. is proud to have conducted a user test on *py-sdk*, the Python version of this new library. Our aim was to provide fresh insights from actual users of the SDK to help build a new generation of BSV wallet infrastructure and offer valuable feedback to improve the SDK's quality. The *py-sdk* development team, led by Script Inc., is notable for its work in building a smart contract platform. The BSV Association engaged Yenpoint Inc. to conduct this user test on *py-sdk*.

The user test was conducted between July 22nd and August 16th, 2024, over a span of three weeks, with the involvement of two engineers. We had full access to the source code, documentation, examples, and a communication channel with the development team.

Primary Tests and Studies:

- Comparison of *py-sdk* with its predecessor, *bsvlib*
- User tests on both basic and new functionalities
- Integration of the Python SDK into a production-ready HD wallet
- Comparison of *py-sdk* with the reference library, *ts-sdk*
- Review of documentation and examples
- Study of how to implement SPV wallet functionality using the Python SDK

During our tests, we identified minor issues, including errors, typos, and inconsistencies in the source code and documentation. These findings were shared with the development team and have been addressed to enhance the SDK's overall quality. Please note that this was a general usability test of the SDK, not a security vulnerability assessment.

The key findings and recommendations are below:

Key Findings

Py-sdk: SPV-ready BSV blockchain library

We conducted a comparative study of *py-sdk* and *bsvlib*, its predecessor. *Py-sdk* represents a significant paradigm shift in design philosophy compared to *bsvlib*. While *bsvlib* was engineered for traditional blockchain wallet creation with a focus on simple P2PKH transactions, *py-sdk* is designed to support Bitcoin's evolution into a globally scalable blockchain. *Py-sdk* provides developers with a robust foundation for implementing Simplified Payment Verification (SPV), enabling a more streamlined yet reliable peer-to-peer payment system. It also offers the capability to manage complex and flexible scripts, including smart contracts.

Key updates in *py-sdk* include:

- Granular and precise manual controls over transactions
- Enhanced functionalities for SPV components, including BEEF and Merkle path verification against blockheaders
- Improved handling of complex scripts with script validation and script template capabilities
- Advanced transaction parsing capabilities and transaction format conversions
- Integration with ARC for broadcasting and receiving Merkle paths

Unlike *bsvlib*'s reliance on external API requests for core functions, *py-sdk* is engineered to support more efficient, scalable operations in line with Bitcoin's future growth. *Please find the detailed reports from the appendix.*

Robust basic functionality and new functionalities for SPV

We conducted a user test for basic functionalities of *py-sdk* as well as new functionalities from the perspective of converting our conventional HD wallet to an SPV-ready wallet using this new library. Please note that this was not a security test for vulnerabilities.

User test on the basic functions and new functions:

- Sending BSV, receiving, and creating transactions
- Complex transactions, including adding extra inputs and outputs
- Custom scripts, 1 sat ordinals, creating, sending, and receiving NFTs
- Basic key functions like BIP 32, 39, 44, etc
- BEEF variation, Merkle path, blockheader service, etc

We successfully tested the above functions and didn’t encounter significant problems in usage as wallets. Overall, we were very impressed with the quality and design of *py-sdk*.
During the tests, we encountered small bugs and errors, and the development team reviewed the reports, which are reflected in commit `12dd3e4` and `1a815f6` of the bitcoin-sv/*py-sdk* GitHub repository.

Implementing the SDK in an HD wallet

We successfully converted the BSV library for our production-ready HD wallet from *bsvlib* to *py-sdk*. Some functionalities, such as key generation, are mostly identical, so it didn't cost much to convert to *py-sdk*. For building transactions, there are notable differences between the two libraries, such as adding fees, inputs and outputs, handling of UTXOs, broadcasting, etc. These parts required additional work and tests. Overall, we were satisfied with the development experience. The usability of the library is much higher, and new functionalities help us build more complex transactions like NFTs.

Three Unified BSV libraries

We conducted a study comparing *py-sdk* with *ts-sdk*, the reference SDK library. We found that the Python SDK (*py-sdk*) and TypeScript SDK (*ts-sdk*) share numerous similarities in their core functionalities. Both were developed using the same set of standards. While there are slight differences in implementation and feature availability, these are not significant. *Please find the detailed reports from the appendix.* However, we did identify a few minor issues, which are reflected in commit `12dd3e4` and `6b4cfac` of the bitcoin-sv/*py-sdk* GitHub repository.

The go-to documentation center for learning BSV development

We found the BSV Skills Center, the official comprehensive documentation site, very helpful for conducting our test. This documentation site offers not only guides for the SDKs but also educational material about protocols, network access rules, important concepts, and network topology, etc. This will be the go-to place for any developers interested in learning these scalable blockchain technologies.

We conducted a review of example codes of *py-sdk* and its documentation. We encountered a few inconsistencies in *py-sdk*, which are reflected in commit `12dd3e4` of the bitcoin-sv/*py-sdk* GitHub repository and in commit `e7d4946b623664b681fda46c7893866acdad6706` of the bitcoin-sv/bsv-skills-center GitHub repository

Study implementing SPV functionalities using py-sdk

The *py-sdk* provides fundamental functionalities for developers to build Simplified Payment Verification (SPV) wallets. While additional development is required, such as implementing a Paymail server using the TypeScript Paymail library or connecting to a pre-built SPV wallet Go server, *py-sdk* addresses the most challenging aspects of SPV development. This report examines the process of creating an SPV wallet using *py-sdk* and explores various implementation options. *Please find the detailed reports from the appendix.*

Conclusions

The BSV Association's development of libraries for two of the most popular languages—Python and JavaScript (with TypeScript as a superset)—as well as Go, one of the most efficient modern languages, is a significant achievement. It will be one of the most important milestones in Bitcoin development. These new SDKs will give much more power to developers to create blockchain applications for the masses. These libraries adhere to the same standard and offer nearly identical functionalities. These well-maintained, unified libraries can significantly reduce language dependency issues and shorten development time for SPV with less confusion. This approach can welcome a wider range of application developers to BSV.

Recommendations for the future development

1. **Increase code comments:** Enhance codebase comments to better explain the purpose of functions, complementing the documentation.

2. **Provide simple examples and basic guides:** Offer more straightforward examples and basic use case guides to help beginners grasp core concepts.

3. **Unify ASM format and version numbers:** Standardize the ASM format and versioning across libraries. *Please find the detailed reports from the appendix.*

4. **Develop a Python Paymail library:** Create a Paymail library for a complete Python development environment. *Please find the detailed report in Implementing an SPV Wallet Using py-sdk from the appendix.*

5. **Expand testing on critical functions:** Conduct further tests on essential functionalities like `transaction.verify` and `spend.validate` to improve SPV validation confidence.

Appendix: *Detailed reports can be found below.*

- Report: A Comparative Analysis of py-sdk and bsvlib
- Report: Comparative Analysis of Python SDK and TypeScript SDK
- Report: ASM Format Comparison Study
- Report: Implementing an SPV Wallet Using py-sdk

Detailed Report:

A Comparative Analysis of *py-sdk* and *bsvlib*

Introduction:

We conducted a comprehensive study comparing *py-sdk* and its predecessor, *bsvlib*. As Yenpoint has been utilizing *bsvlib* for our wallet infrastructure, we are in the process of transitioning to *py-sdk*. This analysis examines the functional differences and design philosophies of both libraries, with a focus on their intended use cases.

Key Findings:

Our investigation revealed fundamental differences in the design concepts of each library. *Bsvlib* was engineered for traditional blockchain wallet creation, whereas *py-sdk* is designed to address a new paradigm: scalable yet simple, embracing the Simplified Payment Verification (SPV) approach.

Bsvlib: Traditional Wallet Infrastructure

Bsvlib prioritizes the facilitation of simple pay-to-public-key-hash (P2PKH) transactions, Bitcoin address transactions. Its design assumes a model where a sender transmits transactions to the blockchain network, and a recipient queries the network via connected full nodes or blockchain API services. *Bsvlib* integrates with blockchain API services, such as What's on Chain, for transaction-related inquiries.

This design philosophy results in a heavy reliance on external API requests for core functions such as receiving transactions, referencing unspent transaction outputs (UTXOs), and broadcasting transactions. While this approach simplifies basic transactions, it presents scalability challenges as user bases expand. The frequent API calls can lead to reduced performance and user experience degradation.

Moreover, while *bsvlib* excels in simplifying basic transactions, it becomes increasingly complex when handling advanced use cases such as non-fungible tokens (NFTs) or fee coverage mechanisms. Such scenarios often necessitate customization of the library itself, reducing flexibility and increasing development overhead.

Py-sdk: A New Era of Bitcoin Development

Py-sdk, while based on *bsvlib*, represents a paradigm shift in design philosophy. It is engineered to support Bitcoin's evolution into a globally scalable blockchain. *Py-sdk* provides developers with a robust foundation for implementing SPV, enabling a more streamlined yet

reliable peer-to-peer payment system, while also offering the capability to manage complex and flexible scripts, including smart contracts.

As blockchain technology scales to mass adoption, the traditional approach of running full nodes or relying on global index services becomes economically unfeasible. Service providers increasingly need to process a growing number of transactions unrelated to their specific applications or user bases, leading to a "Tragedy of the Commons" scenario.

Satoshi Nakamoto proposed SPV as a solution to these scaling challenges in the original Bitcoin whitepaper. However, SPV has not been correctly implemented as Nakamoto envisioned throughout Bitcoin's history. Bitcoin SV is addressing this unresolved blockchain issue, aligning with Satoshi's original vision.

SPV: Simplified Payment Verification

SPV offers an elegant solution to scaling issues through a peer-to-peer payment scheme. In this model, the sender transmits transactions directly to the receiver, who then verifies the transaction and broadcasts it to the network. This approach significantly reduces network queries, as wallets directly receive their transactions, eliminating the need to constantly poll the network for updates.

Furthermore, receivers perform simple verification on incoming transactions using SPV checks, which include source transactions and Merkle paths. This process reduces the propagation of malformed transactions across the network, serving as a first line of defense – analogous to frontend sanitization in general IT development.

Py-sdk: Foundational SPV Functionality

Py-sdk provides fundamental functionalities for developers to build SPV wallets. While additional development is required, such as implementing a Paymail server using the TypeScript Paymail library or connecting to a pre-built SPV wallet Go server, *py-sdk* addresses the most challenging aspects of SPV development.

The library offers granular and precise manual controls over transactions, functionalities previously available only in full node implementations. Once developers grasp the basics of Bitcoin transactions, they can confidently manipulate transactions as needed.

Key Features of *py-sdk*:

1. **Transaction Building:**
    - Utilizes the Transaction() constructor for creating new transaction objects.
    - Employs add_input() and add_output() methods for specifying UTXOs and recipient details.
    - Automates fee calculation and change allocation via the fee() method.
    - Supports change outputs with the change=True flag.
    - Facilitates network broadcasting through the broadcast() method via ARC.

    - Offers serialization options like to_beef() and to_hex() for various transaction formats.
2. **SPV Components:**
    - Implements BEEF (Background Evaluation Extended Format) for SPV transactions.
    - Incorporates Merkle path functionality for transaction inclusion proofs.
3. **Verification:**
    - Provides comprehensive SPV checks through Transaction.verify().
    - Offers specific validations like merkle_path.verify() for Merkle proofs and Spend.validate() for script validations.
4. **Complex Script Handling:**
    - Supports conversion of raw hexadecimal transactions to transaction objects using transaction.from_hex().
    - Introduces a new script template system for generating and customizing scripts.
    - Enables script execution via Spend.validate().

Conclusion:

Py-sdk represents a significant advancement in BSV library development for Python developers, particularly those in AI, IoT, big data, academia, and financial technology. It opens up new possibilities for the future of Bitcoin development, providing a comprehensive toolkit for building scalable, efficient, and sophisticated blockchain applications across these diverse domains. As the Bitcoin ecosystem continues to evolve, *py-sdk* stands as a robust foundation for the next generation of blockchain innovation, enabling developers, researchers, data scientists, and fintech professionals to leverage the power of BSV in their Python projects.

Comparative Analysis of Python SDK and TypeScript SDK

Overview

The comparative analysis reveals that the Python SDK (*py-sdk*) and TypeScript SDK (*ts-sdk*) share numerous similarities in their core functionalities. Both were developed using the same set of standards. While there are slight differences in implementation and feature availability, these are not significant. However, we did identify a few minor issues

Key Similarities

- **Transaction Processing:** Both SDKs offer similar methods for creating, signing, broadcasting, and verifying transactions.
- **Script Processing:** Core script operations, including creation and various conversions, are similarly implemented in both SDKs.
- **SPV Operations:** Both SDKs provide comparable functionality for SPV verification on BEEF, Merkle path, and chaintracker capability.

Notable Differences:

- **Input/Output Handling:** The TypeScript SDK only supports adding single inputs/outputs, while the Python SDK allows for adding multiple inputs/outputs at once
- **Inconsistency in the ASM format:** Both SDKs demonstrate the mostly consistency in converting basic opcodes, but we found a few inconsistencies between the two SDKs—TypeScript and Python.
  See more detail at Report: ASM Format Comparison Study.
- **Errors**: We found an inconsistency behavior / bug on source_txid for tx input. See more detail at `Inconsistent behavior on source_txid`, which are reflected in commit `6b4cfac` of the bitcoin-sv/*py-sdk* GitHub repository.

Conclusion

The BSV Association's development of libraries for two of the most popular languages—Python and JavaScript (with TypeScript as a superset)—as well as Go, one of the most efficient modern languages, is a significant achievement. These libraries adhere to the same standard and offer nearly identical functionalities. These well-maintained, unified libraries can significantly reduce language dependency issues and shorten development

time for SPV with less confusions. This approach can welcome a wider range of application developers to BSV.

Python SDK vs TypeScript SDK Core Feature Comparison Table

Feature Category	Python SDK (*py-sdk*)	TypeScript SDK (*ts-sdk*)	Notes
Transaction Processing			
Transaction Creation	`Transaction` class	`Transaction` class	Similar in both SDKs
Adding Inputs	`add_input()`, `add_inputs()`	`addInput()`	*ts-sdk* supports single input only
Adding Outputs	`add_output()`, `add_outputs()`	`addOutput()`	*ts-sdk* supports single output only
Fee Calculation	`fee()`	`fee()`	Similar in both SDKs
Signing	`sign()`	`sign()`	Similar in both SDKs
Broadcasting	`broadcast()`	`broadcast()`	Similar in both SDKs
Verification	`verify()`	`verify()`	Similar in both SDKs
Script Processing			
Script Creation	`Script` class	`Script` class	Similar in both SDKs
Conversion from ASM	`from_asm()`	`fromASM()`	Similar in both SDKs
Conversion from Hex	`from_hex()`	`fromHex()`	Similar in both SDKs

Conversion from Binary	-	`fromBinary()`	*ts-sdk* only
Conversion to ASM	`to_asm()`	`toASM()`	Similar in both SDKs
Conversion to Hex	`hex()`	`toHex()`	Similar in both SDKs
Conversion to Binary	`serialize()`	`toBinary()`	Similar in both SDKs
Merkle Path			
Merkle Path Creation	`MerklePath` class	`MerklePath` class	Similar in both SDKs
Root Calculation	`compute_root()`	`computeRoot()`	Similar in both SDKs
Verification	`verify()`	`verify()`	Similar in both SDKs
Combination	`combine()`	`combine()`	Similar in both SDKs
Trimming	`trim()`	`trim()`	Similar in both SDKs
Broadcasting			
Broadcast Processing	`Broadcaster` abstract class	`Broadcaster` interface	Different implementation methods
Success Response	`BroadcastResponse` class	`BroadcastResponse` interface	Similar structure
Failure Response	`BroadcastFailure` class	`BroadcastFailure` interface	Similar structure
Chain Tracker			
Root Verification	`is_valid_root_for_height()`	`isValidRootForHeight()`	Similar in both SDKs

Script Templates			
Locking Script Generation	`lock()`	`lock()`	Similar in both SDKs
Unlocking Script Generation	`unlock()`	`unlock()`	*ts-sdk* returns an object
Utility Functions			
Hex Conversion	`to_hex()`	`toHex()`	Similar in both SDKs
Byte Array Conversion	`to_bytes()`	`toArray()`	*ts-sdk* supports multiple encodings
UTF-8 Conversion	`to_utf8()`	`toUTF8()`	*ts-sdk* has more detailed implementation
Base64 Conversion	`to_base64()`	`toBase64()`	Different implementation methods
Base58 Conversion	`from_base58()`, `to_base58()`	`fromBase58()`, `toBase58()`	Similar in both SDKs
Base58Check Conversion	`to_base58_check()`, `from_base58_check()`	`toBase58Check()`, `fromBase58Check()`	Similar in both SDKs
Byte Stream Operations	`Writer`, `Reader` classes	`Writer`, `Reader` classes	Different implementation methods

Note: This table shows a comparison of major features and does not cover all functions or methods. For detailed differences and specific implementations, please refer to the documentation of each SDK.

ASM Format Comparison Study

The ASM (Assembly) format is a valuable tool for programming Bitcoin Script. However, current ambiguities in the ASM format of Bitcoin opcodes make development experiments more challenging and confusing. In this test, we demonstrate inconsistencies between two SDKs—TypeScript and Python. Finally, we provide recommendations for standardizing the ASM format.

Background

Bitcoin ASM (Assembly) is a human-readable form of Bitcoin Script, the language used in Bitcoin transactions. It translates low-level bytecode into understandable mnemonics, making it easier to read and write complex scripts.

Examples expressing a p2pkh in bytecode and ASM.

Bytecode format**:**

```
76 a9 14 89abcdefabbaabbaabbaabbaabbaabbaabbaabba 88 ac
```

ASM format:

```
OP_DUP
OP_HASH160
89abcdefabbaabbaabbaabbaabbaabbaabbaabba
OP_EQUALVERIFY
OP_CHECKSIG
```

Bitcoin, as designed by Satoshi Nakamoto, has comprehensive programmable capabilities using its native script language, Bitcoin Script. However, historically, Bitcoin Core developers limited the use cases of Bitcoin Script, allowing only specific script templates as valid transactions. This restriction reduced opportunities for developers to program Bitcoin Script.

Bitcoin SV aims to return to Bitcoin's original design, enabling most Bitcoin Scripts and eventually all original ones. It offers powerful programmable capabilities and relatively low costs for running large, complex scripts. As a result, more developers will have the opportunity to program Bitcoin Script themselves. Having a uniform ASM format will make development experiments more smooth.

Recommendation

Ideally, the Bitcoin SV Association should standardize the ASM format and implement it to all the BSV infrastructures, libraries, SDKs, arc, full-node, and Teranode etc. Additionally, implementing version numbers for the Bitcoin script would be a significant improvement.

Since the BSV split, we've experienced several changes, with another update on the horizon. Having a clear version indicator is advantageous; for instance, blockchain explorers could utilize specific versions for historical transactions. I encountered this issue firsthand when my old on-chain art couldn't be displayed on WhatOnChain due to a past change in the OP_RETURN implementation.

While this standardization may require substantial effort and time, **it's crucial that, at minimum, all SDKs (Go, TypeScript, and Python) adopt a unified ASM format**. The Go SDK, with its apparently comprehensive list of ASM—information we've gathered regarding its use in Teranode—might serve as an excellent candidate for the standard.

Bytecodes ←→ ASM ←→ Higher-level languages (e.g., Scrypt)

The conversion between bytecodes, ASM, and higher-level languages should be seamless and consistent. This will empower developers to program in ASM, creating complex script templates using higher-level languages like Scrypt, and precisely define script templates for NFTs, FTs, and smart contracts.

Historically, the ASM format has been poorly defined due to limited use cases and Bitcoin Core's disabling of most opcodes. Bitcoin SV has re-enabled most opcodes and plans to restore all opcodes in the near future.

Consequently, developers will craft increasingly complex smart contract scripts, heightening the need for a well-defined ASM format. This will elevate ASM from an ambiguous pseudo-language to a genuine programming language and foundational tool. This shift will usher in a renaissance for Bitcoin smart contracts

Examples of the Different ASM formats for each repositories

Bitcoin-sv fullnode implementation.
https://github.com/bitcoin-sv/bitcoin-sv/blob/master/src/script/opcodes.h

Py-sdk
https://github.com/bitcoin-sv/py-sdk/blob/master/bsv/constants.py

Typescript SDK
https://github.com/bitcoin-sv/ts-sdk/blob/master/src/script/OP.ts

Go SDK
https://github.com/bitcoin-sv/go-sdk/blob/master/script/opcodes.go

About the test

We conducted a comparative test of the ASM format between *ts-sdk* and *py-sdk*. Our methodology involved a two-way conversion process for a comprehensive set of opcodes:

1. From ASM format to hexadecimal representation
2. From hexadecimal representation back to ASM format

This process was carried out using both libraries for each opcode. We then analyzed and compared the test results from *ts-sdk* and *py-sdk* to identify any discrepancies or inconsistencies in their handling of Bitcoin script opcodes.

The aim of this test was to assess the consistency and accuracy of opcode representations across these two important development tools, highlighting any areas where standardization might be needed.

The test findings

1. **Overall Results:**
    - For most opcodes, conversions between *py-sdk* and *ts-sdk* were consistent.
    - Many opcodes successfully converted from ASM to Hex and back to ASM in both SDKs.

2. **Key Differences and Issues:**

    a) OP_FALSE and OP_0:

    b) PUSHDATA Opcodes:

    c) OP_TRUE and OP_1:

    d) Non-standard Opcodes:

    - Both SDKs convert these to the same hex value (00), but interpret it as OP_0 when converting back.

    - In *py-sdk,* the conversion back from OP_FALSE is marked as failed.

    - *py-sdk* successfully converts OP_PUSHDATA1, OP_PUSHDATA2, and OP_PUSHDATA4. - *ts-sdk* appends additional bytes to these opcodes, marking the conversion as failed.

    - Both SDKs use hex value 51, but interpret it differently when converting back.

    - *py-sdk* interprets as OP_TRUE, while *ts-sdk* interprets as OP_1. - Some non-standard opcodes like OP_SUBSTR and OP_LEFT fail conversion in *py-sdk*.

- *ts-sdk* converts these opcodes but interprets them with different names (e.g., OP_SPLIT, OP_NUM2BIN) when converting back.

3. **Matching Results:** - The majority of standard opcodes (e.g., OP_NOP, arithmetic operations, stack operations) show matching conversion results in both SDKs. - Most NOP opcodes (from OP_NOP1 to OP_NOP73) match perfectly.

4. **Summary:** - Both SDKs demonstrate high consistency in converting basic opcodes, but differences emerge in some special cases (particularly PUSHDATA opcodes and non-standard opcodes). - These differences likely stem from variations in SDK implementations or interpretations of Bitcoin Script specifications.

About the test codes

The script is performing the following tasks:

1. Testing Bitcoin Script Opcodes: The script tests the conversion between Assembly (ASM) representation and hexadecimal (hex) representation of Bitcoin Script opcodes.

2. Comprehensive Opcode Coverage: It includes a comprehensive list of Bitcoin Script opcodes, from OP_FALSE (0x00) to OP_INVALIDOPCODE (0xff).

3. Bidirectional Conversion: For each opcode, the test performs two conversions: a. ASM to Hex: Converts the opcode's ASM representation to its hex value. b. Hex to ASM: Converts the hex value back to its ASM representation.

4. Validation: The script checks if the conversions are successful by comparing: a. The converted hex value with the expected hex value. b. The converted ASM representation with the original ASM representation.

5. Logging: Results of each test are logged to a file named 'opcode_test_results.log'. The log includes:

    - The opcode being tested
    - The result of the ASM to Hex conversion
    - The result of the Hex to ASM conversion
    - Whether each conversion was successful or failed

6. Special Cases: The test includes special cases like OP_FALSE and OP_0 which share the same hex value, and opcodes like OP_PUSHDATA which may require additional handling.

7. Error Handling: The conversion functions (asm_to_hex and hex_to_asm) include basic error handling to deal with unknown opcodes.

8. Automated Testing: The script automates the testing process by iterating through all defined opcodes and performing the conversions and checks for each.

This test is crucial for ensuring that the Bitcoin Script opcode conversions in the *py-sdk* library are accurate and reliable, which is fundamental for correctly parsing and creating Bitcoin transactions and scripts.

The details of the comparison results

Implementing an SPV Wallet Using py-sdk

Abstract

The py-sdk provides fundamental functionalities for developers to build Simplified Payment Verification (SPV) wallets. While additional development is required, such as implementing a Paymail server using the TypeScript Paymail library or connecting to a pre-built SPV wallet Go server, py-sdk addresses the most challenging aspects of SPV development. This report examines the process of creating an SPV wallet using py-sdk and explores various implementation options.

Introduction to SPV

SPV, or Simplified Payment Verification, offers an elegant solution to scaling issues through a peer-to-peer payment scheme. In this model, the sender transmits transactions directly to the receiver, who then verifies the transaction and broadcasts it to the network. This approach significantly reduces network queries, as wallets directly receive their transactions, eliminating the need to constantly poll the network for updates.

Furthermore, receivers perform simple verification on incoming transactions using SPV checks, which include source transactions and Merkle paths. This process reduces the propagation of malformed transactions across the network, serving as a first line of defense – analogous to frontend sanitization in general IT development.

For more information on SPV, visit: SPV Documentation

Key Concepts in SPV

1. SPV Transaction Format (BEEF)

BEEF (Background Evaluation Extended Format) is the transaction format for SPV. It allows wallets to perform simplified verification of receiving transactions before broadcasting them to the network, providing a certain level of confidence in the transaction's validity.

BEEF consists of:

- New Transaction
- Source Transactions
- Merkle Paths
- Metadata

Py-sdk has the capability to build and verify BEEF transactions. For more information on BEEF, visit: BEEF Documentation

2. SPV Check (Source Transaction, Merkle Path)

SPV checks involve verifying that source transactions (inputs of the receiving transaction) are already confirmed in blocks. This ensures that the receiving transaction stems from legitimate transactions. Merkle paths are used to prove that source transactions were already included in a block. Py-sdk has the capability to calculate Merkle paths and verify against block headers. Internal or external block-headers-service can be used to receive the newest block headers from the network.

It's important to note that SPV is not a full validation. Full validation requires checking the UTXO set on the network, but SPV might be sufficient for small and micro transactions.

3. Broadcasting (ARC)

In the SPV scheme, the receiver of a transaction is responsible for broadcasting it to the network. ARC endpoints are used for broadcasting. ARC also provides the Merkle paths of the broadcasted transaction once it is included in blocks. Py-sdk has the capability to broadcast via ARC.

4. Communication with Counterparts (Paymail)

Paymail is a crucial component of the SPV scheme, which py-sdk doesn't cover. Developers need to set up and run a Paymail server capable of handling BEEF transactions.

Paymail is a protocol for Bitcoin SV wallets that replaces complex cryptocurrency addresses with simple, email-like handles (e.g., hello@example.com). Your Paymail server communicates with your counterparts to send and receive transactions.

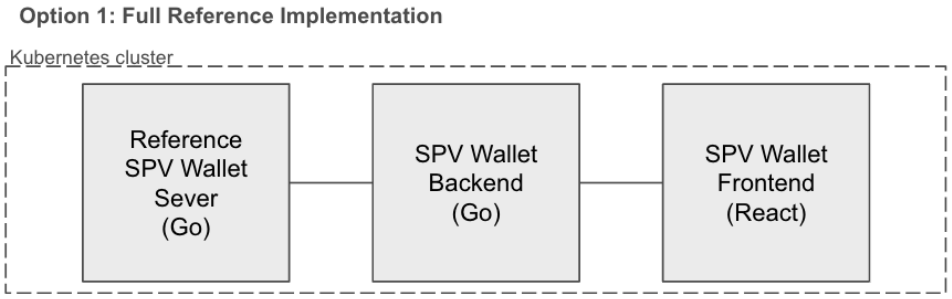
For more information on Paymail, visit: Paymail Documentation

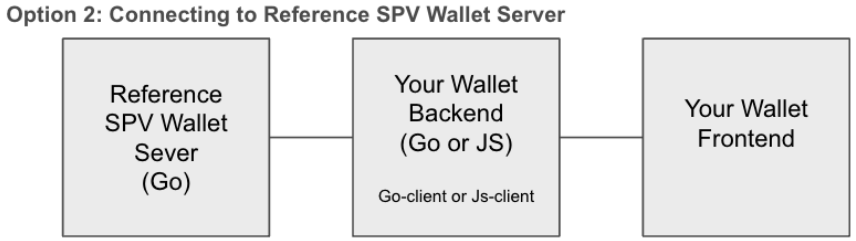
For Paymail BEEF Transaction capability, visit: Paymail BEEF Documentation

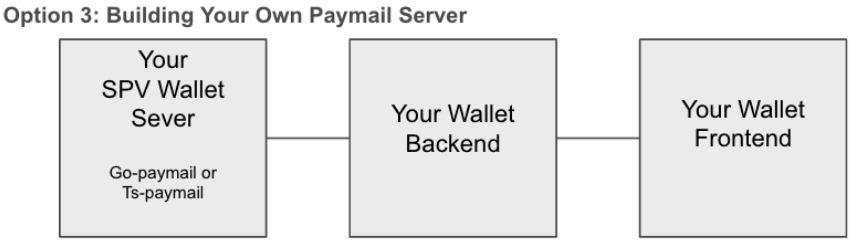
Implementation Options

There are three main options to implement SPV in your wallet:

1. Using the entire reference SPV wallet implementation
2. Connecting a reference SPV wallet server to your wallet backend
3. Building your own Paymail server







Option 1: Full Reference Implementation

The reference implementation SPV wallet, created by the BSV association, is a pre-made, flagship wallet infrastructure. It offers options to deploy a Kubernetes cluster microservice from AWS or your own server. You can customize its wallet backend and frontend for your needs. The downside is the learning curve of running large microservices and the need to understand wallet business logic in Go language for customization.

Option 2: Connecting to Reference SPV Wallet Server

This option allows you to keep your backend and frontend for flexibility while relying on the SPV server for handling Paymail and BEEF transactions. You can connect via API or an SPV library like spv-wallet-go-client or spv-wallet-ts-client. For Python-based wallet backends, you'd need to convert either the Go client or TypeScript client to Python.

Option 3: Building Your Own Paymail Server

This is the most flexible option and can utilize the power of py-sdk for handling BEEF and SPV validation. You can use Paymail libraries such as go-paymail and ts-paymail, potentially converting them to Python. This option requires deep knowledge of the Paymail protocol and effort to build a custom Paymail server.

Option 2: Connecting to Reference SPV Wallet Server for Python environment

Option 3: Building Your Own Paymail Server for Python environment

Conclusion

There is no one-size-fits-all answer for implementing SPV. Developers have multiple options and should choose based on their needs and skill sets.

Key Components

- py-sdk
- spv-wallet
- spv-wallet-go-client
- go-paymail
- ts-paymail
- ARC
- block-headers-service

For more information about BSV and SPV, visit: BSV Documentation